

Project – 12

Node.js



Submitted By:
Nishigandha Patil

INDEX

Sr. No	Topics	Page. No
1	Introduction to Node.js	3
2	Modules	5
3	NPM	8
4	Error Handling	11
5	Asynchronous Programming	14
6	Working with Files	16
7	Command-line Apps	20
8	Working with APIs (Express.js, JWT)	26
9	Working with Database	38
10	Testing, logging & Threads	40

INTRODUCTION TO NODE.JS

➤ What is Node.js?

- Node.js is an open-source, cross-platform JavaScript runtime environment that allows developers to execute server-side JavaScript code.
- It is built on the V8 JavaScript engine, which is the same engine used by Google Chrome, making it fast and efficient.
- It is commonly used for building server-side applications, APIs, and real-time applications, providing a unified language (JavaScript) for both client and server development.

➤ Why Node.js?

1. **Full Stack JavaScript Development:** With Node.js, developers can use JavaScript for both client-side (frontend) and server-side (backend) development.
2. **Asynchronous, non-blocking I/O:** Node.js uses an event-driven, non-blocking I/O model, making it well-suited for handling a large number of concurrent connections without blocking any operation.
3. **Single-Threaded, Event Loop:** Node.js uses a single-threaded event loop to handle multiple client requests concurrently.
4. **Fast Execution:** Node.js is built on the V8 JavaScript engine, which compiles JavaScript code directly into machine code for faster execution.
5. **NPM (Node Package Manager):** NPM is a powerful package manager that allows developers to easily manage dependencies, share code, and leverage a vast ecosystem of third-party libraries and modules.
6. **Modules:** Node.js comes with a built-in module system that allows developers to easily create and reuse modules of code. This allows developers to write modular, reusable code that can be easily shared and maintained.

➤ History of Node.js

Node.js was created by **Ryan Dahl** and was initially released in 2009.

2009: Initial Release

2010: Introduction of npm Package Manager

2011: Version 0.4.0, Node.js and Joyent collaboration

2015: Formation of Node.js Foundation, Introduction of LTS and Node.js and io.js Reconciliation

2016: Node.js 6 LTS, Introduction of Promises

2017: Node.js 8 LTS

2018: Node.js 10 LTS

2019: Node.js 12 LTS

2020: Node.js 14 LTS

2021: Node.js 16 LTS

2022: Node.js V18.0(Active LTS)

2023: Node.js V20.0

2024(Current): Node.js V20.11.1

➤ **Node.js Vs Browser:**

Both browser and Node.js use JavaScript, there are key differences in their ecosystems. Node.js offers the advantage of programming both frontend and backend in a single language, but the environment and APIs differ. In the browser, interaction is with the DOM, while Node.js provides modules like filesystem access. Node.js allows control over the environment, enabling the use of modern JavaScript features without compatibility concerns. Additionally, Node.js supports both CommonJS and ES Modules, whereas the browser is gradually adopting ES Modules.

➤ **The V8 JavaScript Engine:**

V8 is the name of the JavaScript engine that powers Google Chrome. It's the thing that takes our JavaScript and executes it while browsing with Chrome. V8 is written in C++, and it's continuously improved. It is portable and runs on Mac, Windows, Linux and several other systems.

Other JS engines: Other browsers have their own JavaScript engine:

- Firefox has 'SpiderMonkey'
- Safari has 'JavaScriptCore' (also called Nitro)
- Edge was originally based on 'Chakra' but has more recently been rebuilt using 'Chromium' and the V8 engine.

➤ **Installations:**

Go to <https://nodejs.org/en/download> → Download latest version (LTS) → Run the installer.

➤ **Running the node file:**

Open VS Code → Write Node.js code → Save it with <file_name>.js → Open terminal → Use command: `node <file_name.js>` e.g. **node index.js**

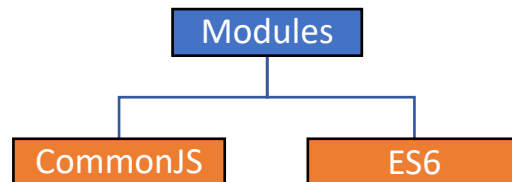
To initialize project as NODE Project use command: **npm init**

It will create **package.json** file.

MODULES

Modules are a fundamental concept that allows you to organize your code into separate files and reuse code across different parts of your application.

Node.js supports a module system called **CommonJS** modules by default. Later, it supports **ES modules** starting in version 14.0.0.



1. CommonJS Modules:

To include a commonJS module, use the **require()** function with the name of the module.

Example: http module (built-in) - `const http = require('node:http')`

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

```
server.js > ...
1  const http = require('node:http');
2  const hostname = '127.0.0.1';
3  const port = 3000;
4  const server = http.createServer((req, res) => {
5    res.statusCode = 200;
6    res.setHeader('Content-Type', 'text/plain');
7    res.end('Hello World\n');
8  });
9  server.listen(port, hostname, () => {
10   console.log(`Server running at http://${hostname}:${port}/`);
11 });
```

- **Create own module using CommonJS:**

Step 1:

Create a file: mymodule.js

Write your JS Code. Use the **module.exports** to make methods and functions available outside the module file.

```
mymodule.js > ...
1  const pi= 3.14;
2  const areaCircle = (r)=>{
3    return pi*r*r;
4  }
5
6  const circumference = (r)=>{
7    return 2*pi*r
8  }
9
10 module.exports = {
11   areaCircle,
12   circumference
13 }
```

Another way to export module:

```
module.exports = areaCircle;
module.exports = circumference;
```

Step 2:

Create another file: main.js

To make the code more concise, you can use the **object destructuring** `{}` when importing a module. We use `./` to locate the module, that means that the module is located in the same folder as the Node.js file.

```
main.js > ...
1  const {areaCircle,circumference} = require('./mymodule')
2
3  console.log("The Area of Circle is:",areaCircle(3))
4  console.log("The Circumference of Circle is:",circumference(3))
5
6
```

Another way to import module:

```
const areaCircle = require('./mymodule')
const circumference = require('./mymodule')
```

Step 3: Run the File: - node main.js

```
PS D:\NODE JS> node main.js
The Area of Circle is: 28.259999999999998
The Circumference of Circle is: 18.84
```

2. ES Modules:

In ES6 modules, you can use the **import** and **export** statements to define dependencies and expose functionality across files.

Note: It's important to note that when working with ES6 modules, you need to use file extensions like `.mjs` for module files or set `"type": "module"` in your 'package.json'.

- **Export/Import module using .mjs:**

You can use the **export** keyword to export variables, functions, or classes from a module.

```
myES6Modules.mjs > ...
1  export const add = (a, b) => a + b;
2  export const subtract = (a, b) => a - b;
3
```

Import module:

```
mainES6.mjs
1  import {add,subtract} from './myES6Modules.mjs';
2
3  console.log("Addition is:",add(10,3));
4  console.log("Subtraction is:",subtract(20,6));
5
```

- **Export/Import Module using .js:**

Save the files with `.js` extension → Go to `package.json` → write `"type": "module"`

```
package.json > ...
1  {
2    "name": "nishu",
3    "version": "1.0.0",
4    "type": "module",
5    "description": "",
6    "main": "index.js",
```

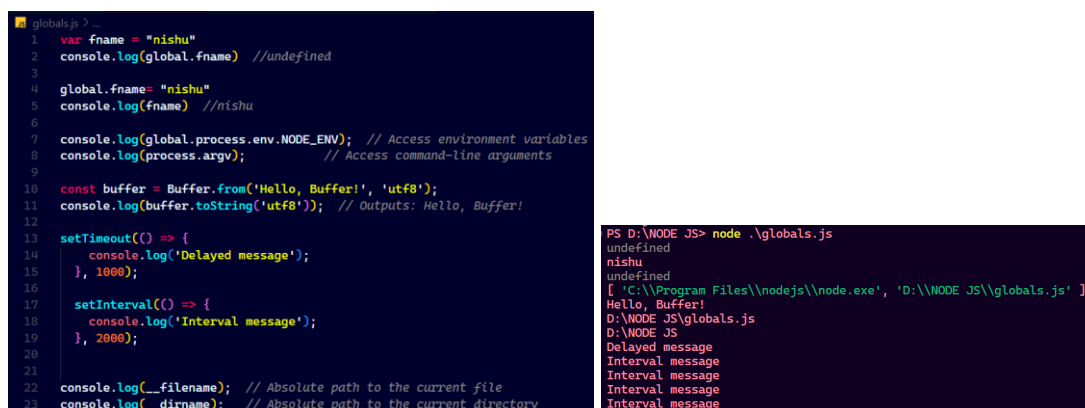
➤ Node.js Build-in Modules:

1. **http:** provides functionality to create HTTP servers and make HTTP requests.
2. **fs:** offers file system-related operations, such as reading and
3. **path:** helps in working with file and directory paths.
4. **os:** provides information about the operating system.
5. **events:** allows the creation and handling of custom events.
6. **util:** provides utility functions for formatting, debugging, and more.
7. **stream:** implements the stream interface for reading or writing data.
8. **url:** helps in parsing and formatting URLs.
9. **net:** offers networking capabilities, including creating TCP servers and clients.
10. **https:** similar to the http module but for creating HTTPS servers.
11. **console:** to log information in the console.
12. **assert:** provides a set of assertion tests.
13. **process:** provides information about, and control over, the current process.
14. **cluster:** able to creating child processes that runs simultaneously and share the same server port.
15. **perf_hooks:** provides APIs for performance measurement.
16. **crypto:** to handle OpenSSL cryptographic functions.
17. **Buffer:** provides APIs to handling streams of binary data.

➤ Globals:

In Node.js, global variables and objects are available throughout the entire application. However, it's essential to use them judiciously, as polluting the global namespace can lead to naming conflicts and unintended consequences.

1. **Global object:** Provides global properties that are available in all parts of a program.
2. **Process object:** Represents the current running process.
3. **Console object:** used for logging and debugging in JavaScript.
4. **Buffer class:** used to handle binary data directly.
5. **setTimeout() and setInterval():** used for asynchronous programming.
6. **__filename:** represents the absolute path of the current module file.
7. **__dirname:** represents the absolute path of the directory containing the current module file



The image shows a code editor on the left and a terminal window on the right. The code in the editor defines a variable 'fname' as 'nishu', logs it, then updates it to 'nishu' and logs it again. It also logs environment variables, command-line arguments, and uses Buffer to create and log a message. It uses setTimeout and setInterval to log 'Delayed message' and 'Interval message' respectively. Finally, it logs __filename and __dirname. The terminal output shows the execution of these statements, resulting in the same log messages as the code.

```
globals.js > ...
1 var fname = "nishu"
2 console.log(global.fname) //undefined
3
4 global.fname= "nishu"
5 console.log(fname) //nishu
6
7 console.log(global.process.env.NODE_ENV); // Access environment variables
8 console.log(process.argv); // Access command-line arguments
9
10 const buffer = Buffer.from('Hello, Buffer!', 'utf8');
11 console.log(buffer.toString('utf8')); // Outputs: Hello, Buffer!
12
13 setTimeout(() => {
14   console.log('Delayed message');
15 }, 1000);
16
17 setInterval(() => {
18   console.log('Interval message');
19 }, 2000);
20
21
22 console.log(__filename); // Absolute path to the current file
23 console.log(__dirname); // Absolute path to the current directory
```

```
PS D:\NODE JS> node .\globals.js
undefined
nishu
undefined
[ 'C:\\Program Files\\nodejs\\node.exe', 'D:\\NODE JS\\globals.js' ]
Hello, Buffer!
D:\NODE JS\globals.js
D:\NODE JS
Delayed message
Interval message
Interval message
Interval message
Interval message
```

NPM

NPM stands for **Node Package Manager**. It is the default package manager for Node.js. It is used to install, manage, and share packages or libraries of code written in JavaScript.

➤ Commonly used npm commands:

1. **npm init:**

Initializes a new **package.json** file for your project. It prompts you for various configuration details.

2. **npm install:**

Installs the dependencies specified in the package.json file and creates **node_modules** folder.

3. **npm install [package-name]:**

Installs a specific package and adds it to the dependencies in the package.json file.
Example: npm install express.

4. **npm install [package-name] --save-dev:**

Installs a package as a development dependency, adding it to the 'devDependencies' in the package.json file.

Example: npm install nodemon --save-dev

Shorthands:

- **-S:** --save
- **-D:** --save-dev

5. **npm uninstall [package-name]:**

Uninstalls a package and removes it from the dependencies in the package.json file.

6. **npm run [script]:**

Executes a script defined in the scripts section of the package.json file.

7. **npm list:**

Displays a tree of installed packages and their dependencies.

8. **npm search [package-name]:**

Searches the npm registry for packages matching the specified keyword.

9. **npm update:**

Updates the installed packages to their latest versions based on the version constraints specified in the package.json file.

10. **npm outdated:**

Shows a list of installed packages that have newer versions available.

➤ Create packages using NPM:

Before starting, create npm Account.

- **Public Packages:**

Open vs code → Create .js file (app.js) → Write some code → Open terminal → Write commands:

npm init → npm login → npm publish --access=public

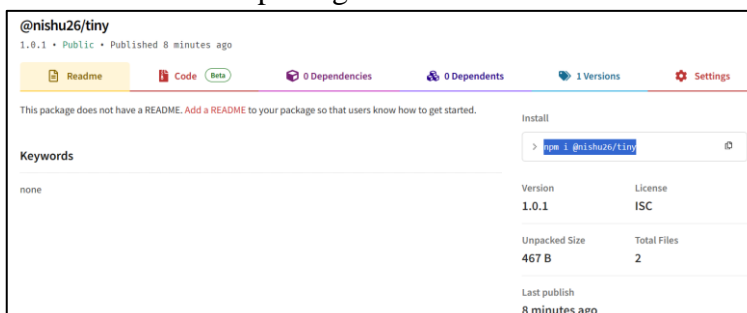
Note: Make sure to update the version in your package.json file before publishing if you are making changes to an existing package.

```
PS D:\New folder\tiny> npm publish --access=public
npm notice
npm notice 📦 @nishu26/tiny@1.0.1
npm notice === Tarball Contents ===
npm notice 258B app.js
npm notice 209B package.json
npm notice === Tarball Details ===
npm notice name: @nishu26/tiny
npm notice version: 1.0.1
npm notice filename: nishu26-tiny-1.0.1.tgz
npm notice package size: 381 B
npm notice unpacked size: 467 B
npm notice shasum: 1443f9595961bc7d101baa2abc4a50e2afced3f4
npm notice integrity: sha512-4BmRqI+17uSpG[...]rErhSswI4kc8Q==
npm notice total files: 2
npm notice Publishing to https://registry.npmjs.org/ with tag latest and public access
* @nishu26/tiny@1.0.1
```

- **Accessing package:**

Users can install and use your public package using the standard **npm install** command.

Login npm → Go to profile → Copy the Installation String → Paste it in your project terminal to use that package.



➤ Installing packages using npm:

In Node.js, packages can be installed either locally or globally, and the scope of their accessibility depends on where they are installed.

1. **Local packages:**

Local packages are accessible only within the directory where they are installed. They are usually listed as dependencies in your project's package.json file.

cd your-project-directory

npm install package-name

2. **Global packages:**

Global packages are accessible from any directory in your terminal. They are available system-wide. They are not listed as dependencies in your project's package.json.

npm install -g package-name

➤ NPX:

npx is a tool that comes with npm, but it is used for executing Node.js packages or commands without having to install them globally or locally.

In summary, **npm** is primarily focused on package management, including installation, versioning, and dependency management, **npx** is a tool for executing Node.js packages and commands in a more flexible and temporary manner, without the need for explicit installation.

➤ Run scripts using npm:

- npm scripts are used for the purpose of initiating a server, starting the build of a project, and also for running the tests.
- We can define these scripts in the package.json file. Also, we can split the huge scripts into many smaller parts if it is needed.

package.json

```
1 {
2   "name": "nishu",
3   "version": "1.0.0",
4   "description": "",
5   "main": "index.js",
6   "scripts": {
7     "test": "echo \"Error: no test specified\" && exit 1"
8   },
9   "author": "nishu",
10  "license": "ISC",
11  "dependencies": {
12    "@nishu26/tiny": "^1.0.1",
13    "express": "^4.18.2",
14    "npm": "^6.0.4",
15    "tiny1": "^1.0.2"
16  },
17 }
```

```
PS D:\NODE JS> npm run test

> nishu@1.0.0 test
> echo "Error: no test specified" && exit 1

"Error: no test specified"
```

npm scripts are used most often for things like starting a server, building the project, and running tests:

```
1 "scripts": {
2   "test": "echo \"Error: no test specified\" && exit 1",
3   "start": "nodemon website.js"
4 },
```

```
PS D:\NODE JS> npm run start

> nishu@1.0.0 start
> nodemon website.js

[nodemon] 3.1.0
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting 'node website.js'
server is listening on port http://localhost:3000
```

➤ nodemon:

Automatically restart the Node.js application when file changes in the directory. This is particularly useful during the development phase.

Install nodemon:

npm install nodemon -g

Run the application:

nodemon <file_name.js> (nodemon app.js)

ERROR HANDLING

Error handling is a crucial aspect of writing robust and reliable Node.js applications. Node.js provides a variety of mechanisms for handling errors.

➤ Types of Errors:

Applications running in Node.js will generally experience four categories of errors:

1. Standard JavaScript errors:

- **SyntaxError:** Occurs when we type code that the JS engine can understand.
- **RangeError:** Occurs when trying to manipulate an object with an invalid length or index.
- **ReferenceError:** Occurs when trying to reference a variable or function that has not been declared or is out of scope.
- **TypeError:** Occurs when a value is not of the expected type.
- **URIError:** URI-Uniform Resource Identifier. These errors typically occur when there's an issue with encoding or decoding URIs.
- **EvalError:** used to identify errors when using the global **eval()** function.

According to EcmaSpec 2018 edition:

This exception is not currently used within this specification.

2. **System errors:** Triggered by underlying operating system constraints such as attempting to open a file that does not exist or attempting to send data over a closed socket.

3. **User-specified errors:** You can create custom errors by extending the built-in Error object.

```
function square(num) {
  if (typeof num !== 'number') {
    throw new TypeError('Expected number but got: ${typeof num}');
  }
  return num * num;
}

try {
  square('8');
} catch (err) {
  console.log(err.message); // Expected number but got: string
}
```

4. **AssertionErrors:** Triggered when Node.js detects an exceptional logic violation that should never occur. These are raised typically by the 'node:assert' module.

```
const assert = require('assert');
Codeium: Refactor | Explain | Generate | JSDoc | X | Codiumate: Options | Test this function
function add(a, b) {
  return a + b;
}

// Assertion: The result of adding 2 and 3 should be 5
try {
  assert.strictEqual(add(2, 3), 8, '2 + 3 should be equal to 5');
  console.log('Assertion passed!');
} catch (error) {
  if (error instanceof assert.AssertionError) {
    console.error('Assertion failed:', error.message);
  } else {
    console.error('Caught an error:', error.message);
  }
}

//Output: Assertion failed: 2 + 3 should be equal to 5
```

➤ Uncaught errors:

Refers to an exception that occurs during the runtime of your application but is not caught by a try-catch block or an error handler.

To handle uncaught exceptions, you can use the 'uncaughtException' event.

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

fs.readFile("somefile.txt", function (err, data) {

  if (err) throw err;

  console.log(data);

});
```

➤ Async errors:

Handling asynchronous errors in Node.js typically involves using mechanisms like try-catch blocks, promises with .catch(), and async/await with try-catch.

```
function asyncFunction() {
  return new Promise((resolve, reject) => {
    // Simulating an asynchronous operation
    setTimeout(() => {
      const success = Math.random() > 0.5;

      if (success) {
        resolve('Operation successful');
      } else {
        reject(new Error('Operation failed'));
      }
    }, 1000);
  });
}

asyncFunction()
  .then(result => {
    console.log(result);
  })
  .catch(error => {
    console.error('Error:', error.message);
  });
```

➤ What is Call Stack?

The call stack is a data structure that keeps track of the function calls in a program. It operates in a Last In, First Out (LIFO) manner, meaning that the most recently called function is the first to be resolved. When a function is called, a new frame is added to the top of the call stack, representing the current function's execution context. When a function completes, its frame is removed from the stack.

➤ Stack Trace:

A stack trace is a textual representation of the call stack at a specific point in time. It provides information about the sequence of function calls that led to an error or an exceptional state. Stack traces are particularly useful for debugging, as they help developers identify the source of an issue.

When an error occurs in your code, Node.js generates a stack trace that is often logged to the console.

➤ Debugging:

Debugging in Node.js can be done using various tools and techniques.

1. Using debugger:

Node.js includes a command-line debugging utility. The Node.js debugger client is not a full-featured debugger, but simple stepping and inspection are possible.

Example - node inspect myscript.js



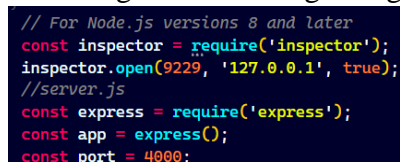
```
PS D:\NodeJS\Edureka\test> node inspect server.js
< Debugger listening on ws://127.0.0.1:9229/3c394fc7-3fdf-4880-b237-d7199232cad7
< For help, see: https://nodejs.org/en/docs/inspector
<
connecting to 127.0.0.1:9229 ... ok
< Debugger attached.
<
Break on start in server.js:1
> 1 const http = require('node:http');
  2 const hostname = '127.0.0.1';
  3 const port = 3000;
debug> next
break in server.js:2
> 1 const http = require('node:http');
> 2 const hostname = '127.0.0.1';
  3 const port = 3000;
  4 const server = http.createServer((req, res) => {
debug> repl
Press Ctrl+C to leave debug repl
> 2+2
4
> .exit
PS D:\NodeJS\Edureka\test>
```

The debugger automatically breaks on the first executable line.

The **repl** command allows code to be evaluated remotely. The next command steps to the next line.

2. Enable Debugging in Your Script:

Make sure your Node.js script has debugging enabled. You can do this by adding the following line at the beginning of your script:



```
// For Node.js versions 8 and later
const inspector = require('inspector');
inspector.open(9229, '127.0.0.1', true);
//server.js
const express = require('express');
const app = express();
const port = 4000;
```

This will open debugger on port 9229

3. Use Chrome DevTools:

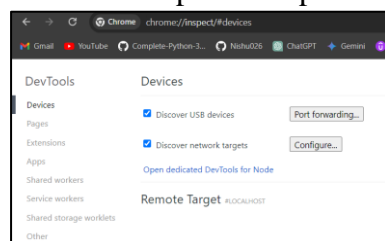
Run the file: node --inspect-brk filename.js

Node.js comes with a built-in debugger that you can access by running your script with the **--inspect** or **--inspect-brk** flag.

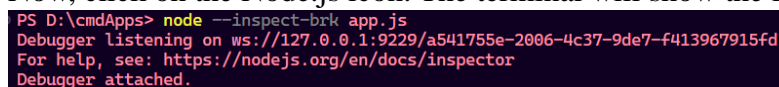
--inspect: run until a breakpoint is encountered.

--inspect-brk: Breaks on the first line, allowing you to step through the code from the beginning.

Open Chrome and type: `chrome://inspect/#devices` → Click on dedicated DevTools for Node → Under the "Remote Target" section, you should see your Node.js script. Click on "inspect" to open DevTools.



Now, click on the Node.js icon. The terminal will show the following message



```
PS D:\cmdApps> node --inspect-brk app.js
Debugger listening on ws://127.0.0.1:9229/a541755e-2006-4c37-9de7-f413967915fd
For help, see: https://nodejs.org/en/docs/inspector
Debugger attached.
```

ASYNCHRONOUS PROGRAMMING

Asynchronous programming allows you to perform non-blocking operations, enabling your applications to handle a large number of concurrent connections efficiently. Node.js uses an event-driven, single-threaded model to achieve asynchronous behaviour.

➤ Event Loop:

- It allows Node.js to be non-blocking and handle concurrent operations efficiently.
- Node.js operates on a single-threaded event loop. This means there is a single process that handles multiple operations concurrently.
- The event loop constantly checks if there are any tasks in the callback queue. If there are, it will execute them one by one.

➤ Event Emitter:

JavaScript in browser provides events such as mouse clicks, keyboard button presses, reacting to mouse movements, and so on.

On the backend side, Node.js offers an option to build a similar system using the **events** module. This module offers the **EventEmitter** class, which we'll use to handle our events. This object has: **on** and **emit** methods.

- **emit** is used to trigger an event.
- **on** is used to add a callback function that's going to be executed when the event is triggered.

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

class MyEmitter extends EventEmitter {}

const myEmitter = new MyEmitter();

myEmitter.on('waterfull', () => {
  console.log('Please Turn OFF the Motor!')
  setTimeout(()=>{
    console.log('Please Turn OFF the Motor!Gentle Reminder.')
  },2000)
});

console.log('The Script is Running!');
myEmitter.emit('waterfull');

// Output:
// The Script is Running!
// Please Turn OFF the Motor!
// Please Turn OFF the Motor!Gentle Remainder.
```

➤ Writing Async Code:

1. **Promise:** It is a proxy for a value not necessarily known when the promise is created. They represent a value that may be available now, in the future, or never. The code either executes or fails, in both cases the subscriber will be notified.

A Promise is in one of these states:

- **pending:** initial state, neither fulfilled nor rejected.
- **fulfilled:** meaning that the operation was completed successfully.
- **rejected:** meaning that the operation failed.

Syntax:

```
let promise = new promise (function (resolve, reject) {  
  //executer  
});
```

2. **async/await:** It is built on top of promises and allows writing asynchronous code that looks and behaves like synchronous code.
The await function works only inside async function. It makes JS wait until the promise settle returns its value.
3. **Callbacks:** A callback is a function called at the completion of a given task; this prevents any blocking, and allows other code to be run in the meantime.
4. **setTimeout:** Runs a function after the specified period expires. Times are declared in milliseconds.
5. **setInterval:** The setInterval() method helps us to repeatedly execute a function after a fixed delay.
6. **setImmediate:** The setImmediate function delays the execution of a function to be called after the current event loops finish all their execution. It's very similar to calling setTimeout with 0 ms delay.
Deprecated: This feature is no longer recommended.
7. **process.nextTick():** Every time the event loop takes a full trip, we call it a tick. When we pass a function to process.nextTick(), we instruct the engine to invoke this function at the end of the current operation before the next event loop tick starts.

- Example of setImmediate, setTimeout and process.nextTick():

```
console.log('Hello => number 1');
setImmediate(() => {
  console.log('Running before the timeout => number 3');
});
setTimeout(() => {
  console.log('The timeout running last => number 4');
}, 0);
process.nextTick(() => {
  console.log('Running at next tick => number 2');
});
```

```
Hello => number 1
Running at next tick => number 2
The timeout running last => number 4
Running before the timeout => number 3
```

The event loop is busy processing the console.log ('Hello => number 1') function code. When this operation ends, the JS engine runs all the functions passed to **nextTick** calls during that operation.

Calling setTimeout(() => {}, 0) will execute the function at the end of next tick.

WORKING WITH FILES

You can programmatically manipulate files in Node.js with the built-in fs module. The name is short for “file system,” and the module contains all the functions you need to read, write, and delete files on the local machine.

1. fs module:

- **Include the 'fs' Module:** Import the 'fs' module at the beginning of your script.
`const fs = require('fs')`

- **Reading File:** Use the `fs.readFile` method:

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

fs.readFile('file.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

The callback is passed (**err**, **data**), where **data** is the contents of the file. If no encoding is specified, then the raw buffer is returned.

- **Writing to a File:** Use the `fs.writeFile` method.

Asynchronously writes data to a file, replacing the file if it already exists.

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

fs.writeFile('outputFile.txt', 'This is a data', 'utf8', (err) => {
  if (err) throw(err)
  console.log('File has been written.');
```

- **Appending File:** Use the `fs.appendFile` method:

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

const contentToAppend = 'This is an appended data.';

fs.appendFile('outputFile.txt', contentToAppend, (err) => {
  if (err) throw err;
  console.log('Data has been appended to the file.');
```

- **Renaming a File:** Use the `fs.rename` method:

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

fs.rename('outputFile.txt', 'myRenamedFile.txt', (err) => {
  if (err) throw err;
  console.log('File has been renamed');
```

- **Deleting a File:** Use the `fs.unlink` method:

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

fs.unlink('myRenamedFile.txt', (err) => {
  if (err) throw err;
  console.log('File has been deleted');
```


- **Checking if a File Exists:** To check if a file exists, you can use the **fs.access** method:

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

fs.access('nishu.txt', fs.constants.F_OK, (err) => {
  if (err) {
    console.error('File does not exist.');//Output: File does not exist.
    return;
  }
  console.log('File exists.');//Output: File exists.
});
```

You can consider using the **fs.promises** API if you prefer working with promises instead of callbacks.

Example:

```
const fs = require('node:fs/promises');
Codeium: Refactor | Explain | Generate JSDoc | X | Codiumate: Options | Test this function
async function readfileFunc() {
  try {
    const data = await fs.readFile('file.txt');
    console.log(data.toString());
  } catch (err) {
    console.log(`Got an error trying to read the file: ${err.message}`);
  }
}
readfileFunc();
```

2. path module:

Include this module in your files using 'const path = require('node:path');'

Given a path, you can extract information out of it using those methods:

- **dirname:** gets the parent folder of a file
- **basename:** gets the filename part
- **extname:** gets the file extension

```
const path = require('node:path');
const notes = '/practise/notes.txt';
console.log(path.dirname(notes)); // /practise
console.log(path.basename(notes)); // notes.txt
console.log(path.extname(notes)); // .txt
```

3. process.cwd() (Current Working Directory):

Returns the current working directory.

```
const { cwd } = require('node:process');

console.log(`Current directory: ${cwd()}`); //Current directory: D:\NODE JS
```

➤ __dirname and __filename:

- **__dirname:** provides absolute path to the directory of the current module.
- **__filename:** provides absolute path to the current module's filename.

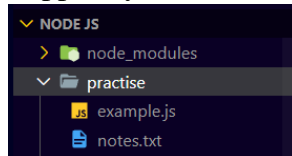
```
console.log('Directory Name:', __dirname) //Directory name:d:\NODE JS\practise
console.log('File Name:', __filename) //File name: d:\NODE JS\practise\example.js
```

➤ **Difference between process.cwd() vs __dirname:**

Use **process.cwd()** when you need the current working directory, which might change during the execution of your script. Use **__dirname** when you need the directory of the current module, and you want a static, unchanging value throughout the script's execution.

Example:

Suppose you have a file, **example.js**, located in the directory **/NODE JS/practise**



If you run your script from that directory:

```
practise > example.js > ...
1  const { cwd } = require('node:process');
2
3  console.log(`Current Working Directory: ${cwd()}`); //Current Working directory: D:\NODE JS\practise
4
5  console.log(`Directory name: '__dirname'`); //Directory name:d:\NODE JS\practise
```

Output:

```
PS D:\NODE JS\practise> node example.js
Current Working directory: D:\NODE JS\practise
Directory name: D:\NODE JS\practise
```

If you change your current working directory using `process.chdir('/some/other/directory')` before accessing these values, `process.cwd()` will reflect the change, but `__dirname` will remain the same.

```
const { cwd } = require('node:process');

process.chdir('/NODE JS');

console.log(`Current Working Directory: ${cwd()}`); //Current Working directory: D:\NODE JS
console.log(`Directory name: '__dirname'`); //Directory name:d:\NODE JS\practise
```

Output:

```
PS D:\NODE JS\practise> node example.js
Current Working Directory: D:\NODE JS
Directory name: D:\NODE JS\practise
```

➤ **Open-Source Packages for File Handling:**

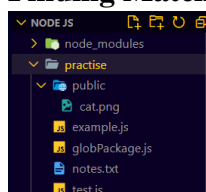
1. **glob:** widely used library for matching files using patterns. It allows you to find files and directories that match a specified pattern based on the rules used by the Unix shell. The patterns are similar to those used in regular expressions but simpler.

Install package: `npm i glob`

Basic Patterns:

- `*`: Matches any number of characters (excluding `/`).
- `?`: Matches a single character (excluding `/`).
- `**`: Matches directories and their subdirectories recursively.

Finding Matches:



```
const {glob} = require('glob')

// Using async/await with glob function
Codeium: Refactor | Explain | X | Codiumate: Options | Test this function
async function findJSFiles() {
  try {
    // Find all JS files recursively, ignoring the node_modules directory
    const jsFiles = await glob('**/*.js', { ignore: 'node_modules/**' });
    console.log('Matching JS files:', jsFiles);

    // Find all png & jpeg recursively in css and public directory
    const images = await glob(['css/*.png,*.jpeg', 'public/*.png,*.jpeg']);
    console.log('Matching JS files:', images);
  } catch (error) {
    console.error('Error:', error);
  }
}

// Call the function to execute the file search
findJSFiles();
```

Output:

```
Matching JS files: [ 'test.js', 'globPackage.js', 'example.js' ]
Matching JS files: [ 'public\\cat.png' ]
```

For more info: <https://www.npmjs.com/package/glob>

2. **globby:** globby is another popular Node.js library for handling file globbing, and it provides a more user-friendly and flexible API compared to the standard glob module. It is built on top of glob but offers additional features and improvements.

For more info: <https://www.npmjs.com/package/globby>

3. **fs-extra:** Enhances the built-in fs module with additional functionality and promise-based versions of its methods.

For more info: <https://www.npmjs.com/package/fs-extra>

4. **chokidar:** chokidar is a Node.js library for watching file and directory changes. It provides a simple and efficient way to monitor file systems for modifications and can be used to trigger various tasks in response to changes, such as rebuilding assets, restarting servers, or updating cached data.

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

chokidar.watch('.').on('all', (event, path) => {
  console.log(event, path);
});
```

```
addDir .
add example.js
add globPackage.js
add notes.txt
add test.js
addDir public
add public\cat.png
change globPackage.js
change globPackage.js
```

For more info: <https://www.npmjs.com/package/chokidar>

COMMAND-LINE APPS

Command Line Applications are applications that can be run from the command line. They are also called CLI (Command Line Interface) applications. Users can interact with clients entirely by terminal commands. They are very useful for automation and building tools.

1. Building a first command-line application:

Step 1: Open VS Code → Create Folder (E.g. cmdApps)

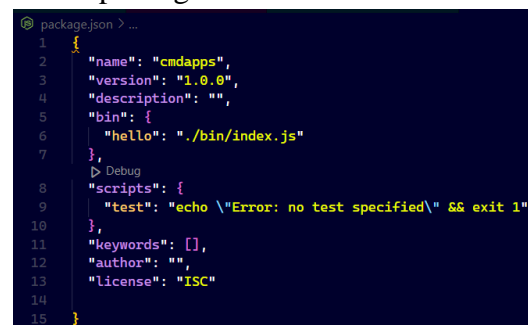
Step 2: Open terminal → npm init

Step 3: Create **bin** folder inside cmdApps → Create **index.js** file inside bin folder.

Step 4: Next, open the **package.json** file in the root of the project in your text editor. Change the **main** value to **bin/index.js**. Add a new key for bin with the following text:

```
"bin": {  
  "hello": "./bin/index.js"  
}
```

Entire package will look like this:

A screenshot of a code editor showing the package.json file. The file contains the following JSON structure: { "name": "cmdapps", "version": "1.0.0", "description": "", "bin": { "hello": "./bin/index.js" }, "scripts": { "test": "echo \\\"Error: no test specified\\\" && exit 1" }, "keywords": [], "author": "", "license": "ISC" }. The file is numbered 1 to 15 on the left margin.

```
1 {  
2   "name": "cmdapps",  
3   "version": "1.0.0",  
4   "description": "",  
5   "bin": {  
6     "hello": "./bin/index.js"  
7   },  
8   "scripts": {  
9     "test": "echo \\\"Error: no test specified\\\" && exit 1"  
10  },  
11   "keywords": [],  
12   "author": "",  
13   "license": "ISC"  
14 }  
15
```

The goal of writing a script like this is to be able to run it from anywhere. Use command:

npm install -g

Step 5: Open cmd and type **“hello”** → It will open the hello script.

Step 6: Install **chalk** and **boxen** packages to modify the color and to add borders to text.

Command: npm install chalk boxen

Step 7: modify the index.js code:

A screenshot of a code editor showing the index.js file. The file contains the following JavaScript code: const chalk = require("chalk"); const boxen = require("boxen"); const greeting = chalk.white.bold("Hello!"); const boxenOptions = { padding: 2, margin: 2, borderStyle: "round", borderColor: "blue", backgroundColor: "#555555" }; const msgBox = boxen(greeting, boxenOptions); console.log(msgBox);. The file is numbered 1 to 15 on the left margin.

```
1 const chalk = require("chalk");  
2 const boxen = require("boxen");  
3  
4 const greeting = chalk.white.bold("Hello!");  
5  
6 const boxenOptions = {  
7   padding: 2,  
8   margin: 2,  
9   borderStyle: "round",  
10  borderColor: "blue",  
11  backgroundColor: "#555555"  
12 };  
13 const msgBox = boxen(greeting, boxenOptions);  
14  
15 console.log(msgBox);
```

Step 8: Install the updated script and run it.

npm install -g .

hello

Output: open cmd > hello

A screenshot of a terminal window showing the command prompt C:\Users\Wishigandha Patil>hello. Below the prompt, a blue-bordered box with a light blue background contains the text "Hello!".

```
C:\Users\Wishigandha Patil>hello  
Hello!
```

2. Command-line Args

- **process.argv:**

The process.argv property returns an array containing the command-line arguments passed when the Node.js process was launched. The first element will be process.execPath. The second element will be the path to the JavaScript file being executed. The remaining elements will be any additional command-line arguments

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

// Printing process.argv property value
var args = process.argv;

console.log("List of arguments:")
args.forEach((val, index) => {
  console.log(`${index}: ${val}`);
});

console.log("Total no. of arguments is: "+args.length);
```

```
PS D:\cmdApps> node index.js
List of arguments:
0: C:\Program Files\nodejs\node.exe
1: D:\cmdApps\index.js
Total no. of arguments is: 2
```

Launching the Node.js process as: node index.js passed_arg1 passed_arg2

```
PS D:\cmdApps> node index.js passed_arg1 passed_arg2
List of arguments:
0: C:\Program Files\nodejs\node.exe
1: D:\cmdApps\index.js
2: passed_arg1
3: passed_arg2
Total no. of arguments is: 4
```

- **Commander.js:**

It is a light-weight, expressive, and powerful command-line framework for node.js. with Commander.js you can create your own command-line interface (CLI).

3. Environment variables:

There are two types of environments: development and production environment. Environment variables in Node.js are a way to store configuration settings outside of your code. They are useful for storing sensitive information, such as API keys, database credentials, or any configuration that might vary between environments (development, testing, production).

In Node.js, you can access environment variables using the 'process.env' object.

- **process. Env:**

It is a global variable that is injected during runtime. When we set an environment variable, it is loaded into process.env during runtime and can later be accessed.

Listing environment variables:

```
console.log(process.env)
```

The process core module that does not require a **require()** method because it is globally available.

Setting Environment Variable:

Example: SET NODE_ENV = "development"/ "production"

4. Exiting and Exit Codes:

Exiting is a way of terminating a Node.js process by using node.js process module.

Here are some common exit codes in Node.js:

Exit Code 0: Success

Exit Code 1: Error

Exit Code 2: Misuse of commands etc.,

Exit Code 3: Internal JavaScript Parse Error

Exit Code 4: Internal JavaScript Evaluation Failure

Exit Code 5: Fatal Error

Exit Code 6: Non-functional Internal Module

- **Exiting of Script Implicitly:** The code will automatically exit the process once it reaches the end and there is nothing to process further.

```
console.log('Code running');
process.on('exit', function(code) {
  return console.log(`exiting the code implicitly ${code}`);
});
```

```
Code running
exiting the code implicitly 0
```

If you modify the above code using setInterval function it will block your Node.js from getting exit.

- **Using process.exit():** The process.exit() is one of the commonly used and quick ways to terminate the process, even if the asynchronous calls are still running.

Syntax:

process.exit(code)

```
console.log('Code running');
process.on('exit', function(code) {
  return console.log(`exiting the code ${code}`);
});
setTimeout((function() {
  console.log("Success");
  return process.exit(0); //node js exit code
}), 3000);
```

```
Code running
Success
exiting the code 0
```

You can also use the **process.exitCode** for terminating the Node.js process safely.

```
process.exitCode = 0 //node js exit code
```

Or simply write:- **process.exit()**

5. Taking Inputs:

Node.js provides a few ways to take inputs from user, including the built-in **process.stdin** and **readline** module. There are also several third-party packages like **prompts** and **Inquirer** built on top of readline that provide an easy to use and intuitive interface.

- **The process.stdin property:** It is an inbuilt application programming interface of the process module which listens for the user input. The stdin property of the process object is a Readable Stream. It uses **on()** function to listen for the event.

Syntax:

process.stdin.on()

```
console.log("Enter Your Name")
process.stdin.on('data', input => {
  console.log(`You Entered: ${input.toString()}`);
  process.exit();
});
```

```
Enter Your Name
Nishu
You Entered: Nishu
```

- **Using prompt-sync package:**

Installation: npm install prompt-sync

```
//prompt-sync
"use strict"
const prompt = require('prompt-sync')();
let name = prompt("Enter Your Name: ")
let age = prompt("Enter Your Age: ")
age = Number.parseInt(age)
console.log(`${name} is ${age} years old.`)
```

```
Enter Your Name: Nishu
Enter Your Age: 23
Nishu is 23 years old.
```

- **Using inquirer package:**

Inquirer.js is a collection of common interactive command line interfaces for taking inputs from user. It is promise based and supports chaining series of prompt questions together, receiving text input, checkboxes, lists of choices and much more.

Installation: npm install inquirer

```
import inquirer from 'inquirer';

inquirer
  .prompt([
    {
      name: 'favColor',
      message: 'What is your favorite color?',
      default: 'red',
    },
    {
      name: 'favDish',
      message: 'What is your favorite dish?',
    }
  ])
  .then(answers => {
    console.info('Answers:', answers);
  });
```

```
node "d:\cmdApps\inquirer.js"
? What is your favorite color? Blue
? What is your favorite dish? Pizza
Answers: { favColor: 'Blue', favDish: 'Pizza' }
```

For more info: <https://www.digitalocean.com/community/tutorials/nodejs-interactive-command-line-prompts>

6. Printing Output:

- **The process.stdout Property:**

It is an inbuilt application programming interface of the process module which is used to send data out of our program. A Writable Stream to stdout. It implements a **write()** method.

```
process.stdout.write('Hello, ');  
process.stdout.write('Node.js!\n');
```

Output: Hello, Node.js

Difference between process.stdout and console.log:

Both stdout and console.log can be used to write messages to the console, console.log is often preferred for its simplicity, automatic newline appending, and ease of use. stdout provides more control but requires additional handling for formatting and newline characters.

- **Chalk package:** It provides an easy way to add colored output to the console, making it helpful for creating visually appealing command-line interfaces (CLIs).

```
const chalk = require('chalk');  
console.log(chalk.blue('Hello, ') + chalk.red('world!'));
```

- **Figlet Package:** Figlet is a package for creating ASCII art text banners using different fonts.

It's often used to add decorative and stylized headers or text to command-line applications.

```
const figlet = require('figlet');  
figlet('Hello, World!', function(err, data) {  
  if (err) {  
    console.log('Something went wrong...');  
    console.dir(err);  
    return;  
  }  
  console.log(data);  
});
```

- **cli-progress:** cli-progress is a library for creating progress bars in the command line interface.

It's useful for showing the progress of long-running tasks, file uploads, or downloads in terminal applications.

```
const cliProgress = require('cli-progress');  
const progressBar = new cliProgress.SingleBar({}, cliProgress.Presets.shades_classic);  
progressBar.start(100, 0);  
for (let i = 0; i <= 100; i++) {  
  progressBar.update(i);  
  // Simulate a delay  
  // Your actual task would go here  
}  
progressBar.stop();
```

Commands:

npm install chalk figlet cli-progress

WORKING WITH APIS

1. API:

➤ What is an API

API stands for Application Programming Interface. It is a set of rules and tools that allows different software applications to communicate with each other. APIs define the methods and data formats that applications can use to request and exchange information.

➤ How do APIs work?

API architecture is usually explained in terms of client and server. The application sending the request is called the client, and the application sending the response is called the server.

There are four different ways that APIs can work depending on when and why they were created.

1. **SOAP APIs:** These APIs use Simple Object Access Protocol. Client and server exchange messages using XML. This is a less flexible API that was more popular in the past.
2. **RPC APIs:** These APIs are called **Remote Procedure Calls**. The client completes a function (or procedure) on the server, and the server sends the output back to the client.
3. **Websocket APIs:** Uses JSON objects to pass data. A WebSocket API supports two-way communication between client apps and the server. The server can send callback messages to connected clients, making it more efficient than REST API.
4. **REST APIs:** REST stands for Representational State Transfer. REST defines a set of functions like GET, PUT, DELETE, etc. that clients can use to access server data. Clients and servers exchange data using HTTP.

The main feature of REST API is statelessness. Statelessness means that servers do not save client data between requests.

➤ What is Web API?

A Web API or Web Service API is an application processing interface between a web server and web browser. All web services are APIs but not all APIs are web services. REST API is a special type of Web API that uses the standard architectural style.

➤ What are API integrations?

API integrations are software components that automatically update data between clients and servers. Some examples of API integrations are when automatic data sync to the cloud from your phone image gallery.

➤ What is an API endpoint and why is it important?

An API endpoint is a specific URL or URI (Uniform Resource Identifier) that represents a specific function or resource in a web-based application. It serves as the access point or entry point for a particular interaction between a client (such as a web browser or another application) and a server. The endpoint defines where and how an API can be accessed and specifies the operations that can be performed on a resource.

API endpoints are critical to enterprises for two main reasons:

1. Security

API endpoints make the system vulnerable to attack. API monitoring is crucial for preventing misuse.

2. Performance

API endpoints, especially high traffic ones, can cause bottlenecks and affect system performance.

➤ **How to secure a REST API?**

All APIs must be secured through proper authentication and monitoring. The two main ways to secure REST APIs include:

- **Authentication tokens:**

These are used to authorize users to make the API call. Authentication tokens check that the users are who they claim to be and that they have access rights for that particular API call. For example, when you log in to your email server, your email client uses authentication tokens for secure access.

- **API keys:**

API keys verify the program or application making the API call. They identify the application and ensure it has the access rights required to make the particular API call. API keys are not as secure as tokens but they allow API monitoring in order to gather data on usage. You may have noticed a long string of characters and numbers in your browser URL when you visit different websites. This string is an API key the website uses to make internal API calls.

➤ **How to Create an API:**

- **Plan the API**

- **Build the API:** API designers prototype APIs using boilerplate code. Once the prototype is tested, developers can customize it to internal specifications.

- **Test the API:** Use Testing Softwares to test the API

- **Document the API:** Good API documentation is crucial. Consider using tools like Swagger/OpenAPI or writing your documentation in Markdown. Describe each endpoint, the expected parameters, and the responses.

- **Deploy the API:** When you are ready, deploy your API to a hosting service.

➤ **The steps to implement a new API include:**

- Obtaining an API key. This is done by creating a verified account with the API provider.

- Set up an HTTP API client. This tool allows you to structure API requests easily using the API keys received.

- If you don't have an API client, you can try to structure the request yourself in your browser by referring to the API documentation.

- Once you are comfortable with the new API syntax, you can start using it in your code.

➤ **Where to find new APIs?**

New web APIs can be found on API marketplaces and API directories. API marketplaces are open platforms where anyone can list an API for sale.

Some popular API websites include:

- **Rapid API** –RapidAPI allows users to test APIs directly on the platform before committing to purchase.
- **Public APIs** – The platform groups remote APIs into 40 niche categories, making it easier to browse and find the right one to meet your needs.
- **APIForThat and APIList** – Both these websites have lists of 500+ web APIs, along with in-depth information on how to use them.

➤ **What is an API gateway?**

An API Gateway is an API management tool for enterprise clients that use a broad range of back-end services. API gateways typically handle common tasks like user authentication, statistics, and rate management that are applicable across all API calls.

➤ **What is GraphQL?**

GraphQL is a query language that was developed specifically for APIs. It prioritizes giving clients exactly the data they request and no more. It is designed to make APIs fast, flexible, and developer-friendly.

2. Http Server:

- **HTTP Module:** It is a built-in module that provides functionality for creating HTTP servers and making HTTP requests.

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

const host = 'localhost';

const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('My First Server'); //Output
});
server.listen(port, host, () => {
  console.log(`Server is running on http://${host}:${port}`);
});
```

Instead of plain text we can set html, json and csv content.

For HTML: res.setHeader(‘Content-Type’, ‘text/html’)

```
const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/html');
  res.end('<h1>Hello World</h1>'); //Output
});
```

For JSON: res.setHeader(‘Content-Type’, ‘application/json’ or ‘text/json’)

```
const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/json');
  res.end(`{"message": "This is a JSON response"}`); //Output
});
```

➤ Express.js Framework:

Express is a Node.js web application framework that provides broad features for building web and mobile applications. It is used to build a single page, multipage, and hybrid web application.

It comes with a default template engine, **Jade** which helps to facilitate the flow of data into a website structure and does support other template engines.

It supports **MVC** (Model-View-Controller), a very common architecture to design web applications.

- **Steps:**

1. **Create 'package.json':** npm init
2. **Installing Express:** npm i express
3. **Building Basic Server:**

```
//server.js
const express = require('express');
const app = express();
const port = 4000;

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

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

4. **Run Server:** node server.js
5. **Output:** Visit <http://localhost:4000> to see output.

- **Basic Routing in Express:** Express uses routes to handle different HTTP requests.

```
//server.js
const express = require('express');
const app = express();
const port = 4000;

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

app.get('/about', (req, res) => {
  res.send('This is an About Page');
});

app.get('/services', (req, res) => {
  res.send('This is a services Page');
});

app.get('/contact-us', (req, res) => {
  res.send('This is a Contact Us Page');
});

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

- **Routing methods:**
 1. **app.get(path, callback [, callback ...]):** Handles GET requests for the specified path.
 2. **app.post(path, callback [, callback ...]):** Handles POST requests for the specified path.
 3. **app.put(path, callback [, callback ...]):** Handles PUT requests for the specified path.
 4. **app.delete(path, callback [, callback ...]):** Handles DELETE requests for the specified path.
 5. **app.all(path, callback [, callback ...]):** Handles all HTTP methods for the specified path.
 6. **app.use([path,] callback [, callback...]):** Mounts middleware functions at the specified path. If no path is specified, the middleware is executed for all paths.

- **Middleware Functions in Express:**

Middleware functions are those functions that have access to the request object (req), the response object (res), and the next function in the application's request-response cycle. The objective of these functions is to modify request and response objects for tasks like parsing request bodies, adding response headers, make other changes to request-response cycle, end the request-response cycle and call the next middleware function.

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

Codeium: Refactor | Explain | Generate JSDoc | X | Codiumate: Options | Test this function
const logMiddleware = (req, res, next) => {
  console.log('New Request Received at: [${new Date().toLocaleString()}] ${req.method} ${req.url}');
  next(); // Call the next middleware in the stack
};

// Use the middleware for all routes
app.use(logMiddleware);
```

Output:

```
PS D:\node> node server.js
Example app listening on port http://localhost:3000
New Request Received at:[6/3/2024, 10:11:37 pm] GET /about
New Request Received at:[6/3/2024, 10:11:44 pm] GET /
New Request Received at:[6/3/2024, 10:11:52 pm] GET /contact
New Request Received at:[6/3/2024, 10:12:27 pm] GET /contact
```

- **Middleware Function for specific route:**

```
//Simple request time logger for a specific route
app.use('/', (req, res, next) => {
  res.send('This is a Home Page');
  console.log('A new request received at ' + Date.now());
  next();
});
```

```
PS D:\node> node server.js
Example app listening on port http://localhost:3000
A new request received at 1709743686482
```

- **Third Party Middleware Functions:**

1. **Body-parser:** body-parser is a middleware for Express.js that extracts the entire body portion of an incoming request and exposes it on the req.body property. It is particularly useful for parsing data from HTML forms or other types of requests where the data is sent in the body of the request.

Installations: `npm install body-parser`

Example:

```
const express = require('express')
const path = require('path')
const bodyParser = require('body-parser');

const app = express()
const port = 3000

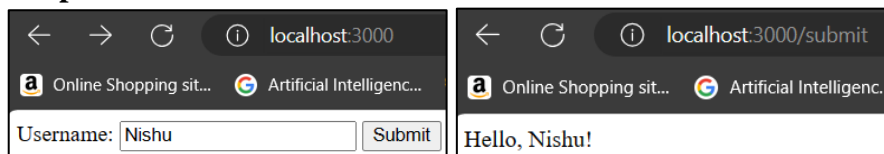
// To parse URL encoded data
app.use(bodyParser.urlencoded({ extended: false }));

// To parse json data
app.use(bodyParser.json());

app.get('/', (req, res) => {
  res.send(`
    <form method="post" action="/submit">
      <label for="username">Username:</label>
      <input type="text" id="username" name="username" required>
      <button type="submit">Submit</button>
    </form>
  `);
});

// Handle form submission
app.post('/submit', (req, res) => {
  const username = req.body.username;
  res.send(`Hello, ${username}!`);
});
```

Output:



2. **cookie-parser:** It parses Cookie header and populate req.cookies with an object keyed by cookie names.

```
const cookieParser = require('cookie-parser');
app.use(cookieParser());
```

3. **session:** A session is often used in applications such as login/signup.

```
app.use(
  session({
    secret: 'arbitrary-string',
    resave: false,
    saveUninitialized: true,
    cookie: { secure: true }
  })
)
```

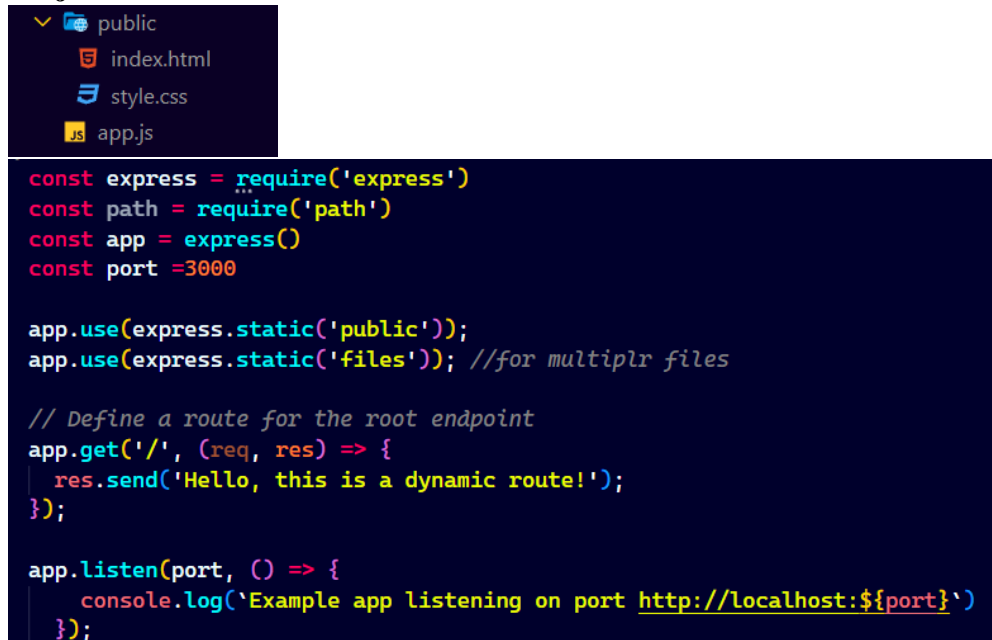
4. **morgan:** keeps track of all the requests and other important information depending on the output format specified.

```
const logger = require('morgan');
// ... Configurations
app.use(logger('common'));
```

- **Serving Static Files:**

Express.js allows you to serve static files, such as HTML, CSS, images, and JavaScript files, with the help of its built-in `express.static` middleware. This middleware makes it easy to serve static content without the need for additional configuration.

Project Structure:



➤ **Templating Engines:**

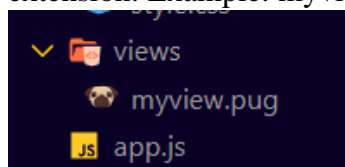
Express.js allows developers to use template engines to generate dynamic HTML content on the server side and then send it to the client. There are several template engines compatible with Express, and each has its own syntax and features.

1. **Pug:** Pug is a concise and expressive template engine with a syntax that uses indentation rather than HTML tags.

Step 1: npm install pug

Step 2: Create express app with **app.js**

Step 2: Create views folder in your project directory. create file called with **.pug** extension: Example: myview.pug



app.js



myview.pug:

```
views > myview.pug
1  doctype html
2  html(lang="en")
3  head
4    meta(charset="UTF-8")
5    meta(name="viewport", content="width=device-width, initial-scale=1.0")
6    title= MyWebpage
7  body
8    h1 Welcome to #{msg}
9    //- msg variable
```

2. **EJS (Embedded JavaScript):** EJS allows you to embed JavaScript code directly in your HTML templates. It has a syntax similar to HTML, making it easy for developers familiar with HTML to get started. In EJS, `<% %>` tags are used for control flow, and `<%= %>` tags are used to output the result of a JavaScript expression.

Installation: npm install ejs

In views folder create a file with `.ejs` extension: **index.ejs**

```
views > index.ejs > ...
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <meta charset="UTF-8">
5    <meta name="viewport" content="width=device-width, initial-scale=1.0">
6    <title><%= title %></title>
7  </head>
8  <body>
9    <h1>Welcome to <%= title %></h1>
10 </body>
11 </html>
```

app.js:

```
app.js > ...
1  const express = require('express')
2  const app = express()
3  const port = 3000
4
5  app.set('view engine', 'ejs')
6  // app.set('views', './views');
7
8  app.get('/', (req, res) => {
9    res.render('index', { title: 'Express with Ejs' });
10 })
11
12 app.listen(port, () => {
13   console.log(`Example app listening on port http://localhost:${port}`)
14 });
```

3. **Marko:** Marko is another popular template engine for building fast and efficient web applications with Node.js and Express. It is developed by eBay and provides a concise and powerful syntax.

In Marko, `${}` syntax is used for variable interpolation.

Installations: npm install marko

In views folder create a file with `.marko` extension: **index.marko**

4. **Handlebars:** Handlebars is a logic-less template engine that allows you to create templates with dynamic content using placeholders.

Installation: npm install express-handlebars

The advantage of using a Template Engine over raw HTML files is that they provide support for performing tasks over data. HTML cannot render data directly. Frameworks like Angular and React share this behaviour with template engines.

➤ Project Structure of an Express App:

```
project-root/
  node_modules/           // This is where the packages installed are stored
  config/
    db.js                 // Database connection and configuration
    credentials.js        // Passwords/API keys for external services used by your app
    config.js             // Environment variables
  models/                 // For mongoose schemas
    books.js
    things.js
  routes/                 // All routes for different entities in different files
    books.js
    things.js
  views/
    index.pug
    404.pug
    ...
  public/                 // All static files
    images/
    css/
    javascript/
  app.js
  routes.js               // Require all routes in this and then require this file in
  app.js
  package.json
```

3. Making API Calls:

API calls: Application Programming Interface calls, refer to the process of making requests to an external server or service to retrieve or send data.

API calls are commonly used in web development to integrate third-party services, access external databases, or interact with other web applications. For example, a front-end web application might make API calls to a back-end server to fetch or update data without reloading the entire page.

Making API calls in Node.js involves using the `http` or `https` module to send HTTP requests to a server. Additionally, you can use third-party libraries like `axios` or `node-fetch` to simplify the process and handle requests more efficiently.

In the context of web development, API calls are often associated with HTTP requests made to a server. There are several types of API calls, with the most common ones being:

- **GET:** Retrieves data from the server.
- **POST:** Submits data to be processed to a specified resource.
- **PUT:** Updates a resource on the server.
- **DELETE:** Deletes a specified resource on the server.

1. **got package:** `got` is a popular HTTP request library for Node.js that simplifies making HTTP requests and handling responses. It provides a more user-friendly interface compared to the built-in `http` module.

npm install got

```
import got from 'got';
Codiumate: Options | Test this function
const getData = async () => {
  try {
    const res = await got
      .get('https://jsonplaceholder.typicode.com/posts/1')
      .json();
    console.log(res);
  } catch (err) {
    console.log(err);
  }
};

getData()
```

Output:

```
{
  userId: 1,
  id: 1,
  title: 'sunt aut facere repellat provident occaecati excepturi optio reprehenderit',
  body: 'quia et suscipit\n' +
    'suscipit recusandae consequuntur expedita et cum\n' +
    'reprehenderit molestiae ut ut quas totam\n' +
    'nostrum rerum est autem sunt rem eveniet architecto'
}
```

2. **axios package:** Axios is a popular HTTP client for making XMLHttpRequests or HTTP requests in Node.js and web browsers. It provides a simple and consistent interface for interacting with APIs.

npm install axios

```
import axios from "axios";

// Replace the URL with the actual API endpoint
const apiUrl = 'https://jsonplaceholder.typicode.com/posts/1';

axios.get(apiUrl)
  .then(response => {
    console.log(response.data); // Output the response data
  })
  .catch(error => {
    console.error(error); // Output any errors that occurred
  });
```

Axios post request:

```
// Send a POST request
axios({
  method: 'post',
  url: '/user/12345',
  data: {
    firstName: 'Fred',
    lastName: 'Flintstone'
  }
});
```

4. Authentication:

- **JWT:**

JWT, or JSON Web Token, is an open standard used to share security information between two parties — a client and a server. Each JWT contains encoded JSON objects, including a set of claims. JWTs are signed using a cryptographic algorithm to ensure that the claims cannot be altered after the token is issued.

- **JSON:**

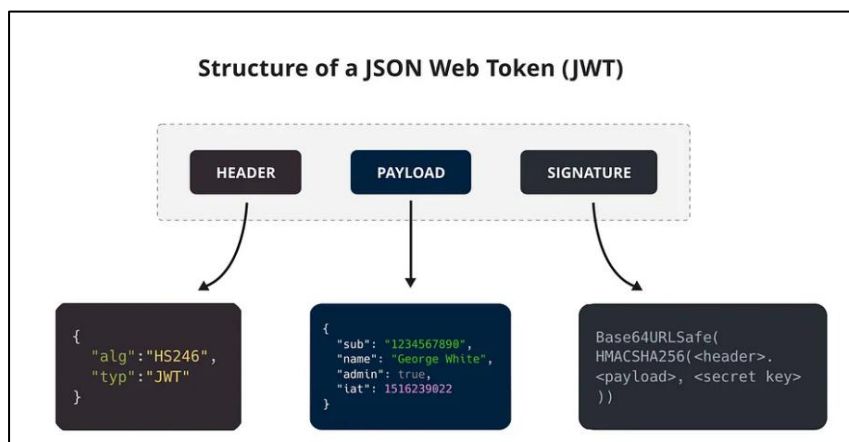
JSON stands for JavaScript Object Notation and is a text-based format for transmitting data across web applications. It stores information in an easy-to-access manner, both for developers and computers. It can be used as a data format by any programming language and is quickly becoming the preferred syntax for APIs

- **TOKEN:**

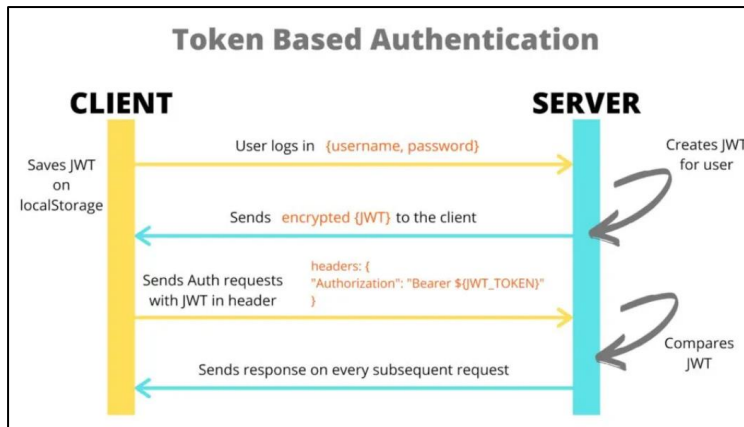
A token is a string of data that represents something else, such as an identity. In the case of authentication, a non-JWT based token is a string of characters that allow the receiver to validate the sender's identity.

- **How JWT Works?**

A JWT is a string made up of three parts, separated by dots (.), and serialized using base64. In the most common serialization format, compact serialization, the JWT looks something like this: xxxxx.yyyyy.zzzzz.



1. **Header:** Consists of two parts — the token type (JWT) and the signing algorithm being used, such as HMAC SHA256 or RSA.
2. **Payload:** Contains the claims, which are statements about the user or other data. Claims can be of three types: registered, public, and private claims.
3. **Signature** The signature is created by combining the encoded header, encoded payload, and a secret key using the specified algorithm. The signature is used to verify that the sender of the JWT is who it says it is and to ensure that the message wasn't changed along the way.



When a user logs in or attempts to access a protected resource, the server generates a JWT after successful authentication. The client then stores this token, usually in local storage or a cookie. For every subsequent request that requires authentication, the client sends the JWT in the request headers. The server validates the token by checking the signature and decoding the payload to ensure the user's authenticity and authorization.

Installation: `npm install jsonwebtoken`

Creating JWT:

```
import jwt from 'jsonwebtoken';

const secretKey = 'Nishu@123456$1234';

const sampleUser = {
  id: '123',
  username: 'Nishu Patil',
  role: 'user'
};

const generateJWT = (user) => jwt.sign(user, secretKey, { expiresIn: '1h' });

Codeium: Refactor | Explain | Generate JSDoc | X | Codiumate: Options | Test this function
const verifyJWT = (token) => {
  try {
    return jwt.verify(token, secretKey);
  } catch (err) {
    console.error('JWT verification failed:', err.message);
    return null;
  }
};

const userToken = generateJWT(sampleUser);
console.log('Generated JWT:', userToken);

const decodedToken = verifyJWT(userToken);
if (decodedToken)
  console.log('Decoded JWT payload:', decodedToken);
```

Output:

```
Generated JWT: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ3ZCI6IjEyMyIsInVzZXJyZWV1IjoiaTlzaHUgUGF0aWw1IjCjY2b2x1IjoidXNlciIsImhhdCI6MTcwOTg3ODYxMSwiZXhwaW90IjozNzA5ODgyMjExfQ.Yh5oi8jmAW7j7YhXvTv0aBmGcqxUr920HIqCd0xukh8
Decoded JWT payload: {
  id: '123',
  username: 'Nishu Patil',
  role: 'user',
  iat: 1709878611,
  exp: 1709882211
}
```

WORKING WITH DATABASE

Working with databases in Node.js often involves interacting with a database management system (DBMS) such as MySQL, PostgreSQL, MongoDB, or others.

1. Document:

A document database is a type of nonrelational database that is designed to store and query data as JSON-like documents.

- **MongoDB:**

mongoose package: npm install mongoose

VS Code Extension: MongoDB For VS Code

Creating schema and Model:

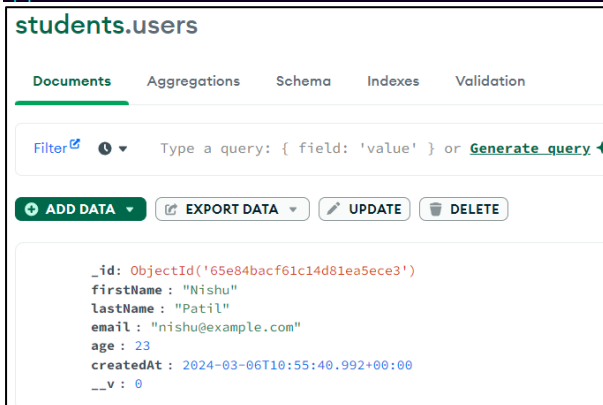
```
userModel.js > ...
1 // mongoose.js
2 const mongoose = require('mongoose');
3 const Schema = mongoose.Schema;
4
5 // Define the Schema
6 const userSchema = new Schema({
7   firstName: {
8     type: String,
9     required: true
10  },
11  lastName: {
12    type: String,
13    required: true
14  },
15  email: {
16    type: String,
17    required: true,
18    unique: true
19  },
20  age: {
21    type: Number,
22    min: 0
23  },
24  createdAt: {
25    type: Date,
26    default: Date.now
27  },
28 });
29
30 // Create a Model
31 const User = mongoose.model('User', userSchema);
32
33 // Export the Model
34 module.exports = User;
```

Inserting data:

```
app.js > ...
1 // app.js or wherever you set up your application
2 const mongoose = require('mongoose');
3 const User = require('./userModel');
4
5 mongoose.connect("mongodb://0.0.0.0:27017/students")
6   .then(() => {
7     console.log('Connected to MongoDB');
8
9     // Now you can use the User model for CRUD operations
10    // For example, create a new user
11    const newUser = new User({
12      firstName: 'Nishu',
13      lastName: 'Patil',
14      email: 'nishu@example.com',
15      age: 23
16    });
17
18    newUser.save()
19      .then(user => {
20        console.log('User created:', user);
21      })
22      .catch(error => {
23        console.error('Error creating user:', error);
24      });
25  })
26  .catch(error => {
27    console.error('MongoDB connection error:', error);
28  });
```

Output:

```
PS D:\ChocolateStay\MongoDrivers> node app.js
Connected to MongoDB
User created: {
  firstName: 'Nishu',
  lastName: 'Patil',
  email: 'nishu@example.com',
  age: 23,
  _id: new ObjectId('65e84bacf61c14d81ea5ece3'),
  createdAt: 2024-03-06T10:55:40.992Z,
  __v: 0
}
```



For More Info: <https://www.mongodb.com/developer/languages/javascript/getting-started-with-mongodb-and-mongoose/>

- **Prisma:**

Prisma is an open source next-generation ORM in the TypeScript ecosystem. It offer a dedicated API for relation filters. It provides an abstraction layer that makes you more productive compared to writing SQL. Prisma currently supports PostgreSQL, MySQL, SQL Server, SQLite, MongoDB and CockroachDB.

2. Relational Database:

- **Knex:** 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.
- **TypeORM:** TypeORM is an Object-Relational Mapping (ORM) library for TypeScript and JavaScript that works with relational databases. TypeORM provides a way to interact with databases using TypeScript classes and decorators.
- **Sequelize:** It is a popular ORM library for Node.js. Sequelize provides a convenient way to interact with databases by abstracting the SQL queries and allowing developers to use JavaScript or TypeScript to work with database models.
- **Prisma:** It is an ORM that helps app developers build faster and make fewer errors. Combined with its Data Platform developers gain reliability and visibility when working with databases.

TESTING, LOGGING & THREADS

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. Testing:

- **Jest (JavaScript Testing Framework):**

Jest is a popular JavaScript testing framework developed by Facebook. It is commonly used for testing React applications, but it can be used for any JavaScript project. Jest provides a testing environment, assertion library, and a powerful set of features for writing and running tests.

Installation: npm install jest

Create **sum.js** file to add two numbers:

```
function sum(a, b) {  
  return a + b;  
}  
  
module.exports = sum;
```

Then, create a file named **sum.test.js**. This will contain our actual test:

```
sum.test.js > ...  
1  const sum = require('./sum')  
2  
   Codiumate: Add more tests  
3  test('adds 1 + 2 to equal 3', () => {  
4    expect(sum(1, 2)).toBe(3)  
5  });
```

Add test script to your package.json:

```
Debug  
"scripts": {  
  "test": "jest"  
},
```

Run: npm test

```
PS D:\cmdApps> npm test  
  
> cmdapps@1.0.0 test  
> jest  
  
PASS ./sum.test.js  
  ✓ adds 1 + 2 to equal 3 (4 ms)  
  
Test Suites: 1 passed, 1 total  
Tests:      1 passed, 1 total  
Snapshots:  0 total  
Time:       0.775 s, estimated 1 s  
Ran all test suites.
```


- **Cypress:**

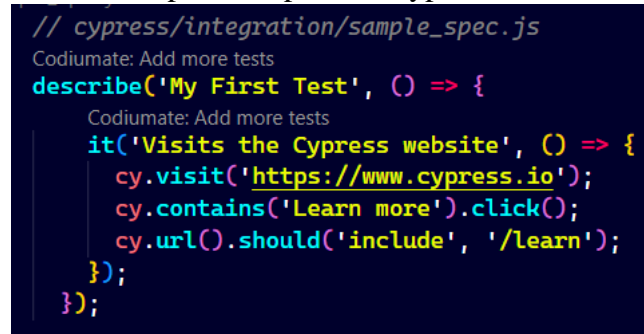
Cypress is a JavaScript end-to-end testing framework designed for modern web applications. It is commonly used to test the functionality of web applications by simulating user interactions within a real browser environment.

Installation: `npm install --save-dev cypress`

After installation, you can open Cypress using the command:

`npx cypress open`

Here's a simple example of a Cypress test:



```
// cypress/integration/sample_spec.js
Codiumate: Add more tests
describe('My First Test', () => {
  Codiumate: Add more tests
  it('Visits the Cypress website', () => {
    cy.visit('https://www.cypress.io');
    cy.contains('Learn more').click();
    cy.url().should('include', '/learn');
  });
});
```

2. Logging:

Logging in the context of software development refers to the practice of recording information about the execution of a program. It is a crucial aspect of debugging, monitoring, and analysing the behaviour of an application. Logging helps developers and administrators understand what happened within the application, identify issues, and track the flow of execution.

➤ Logging Levels:

Logging statements are typically categorized into different levels based on their importance or severity. Common logging levels include:

- **DEBUG:** Detailed information useful for debugging.
- **INFO:** General information about the application's operation.
- **WARN:** Warning messages indicating potential issues.
- **ERROR:** Error messages for unexpected and potentially harmful events.
- **FATAL:** Critical errors that may lead to application termination.

console.log: 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)`

➤ Node.js Logging Packages:

- **Winston package:**

Winston is a versatile logging library for Node.js that provides a simple and extensible way to log messages with different levels, formats, and transports. It's widely used in various Node.js applications for logging events, errors, and other information.

Basic logging with winston:

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

const logger = winston.createLogger({
  level: 'info',
  format: winston.format.simple(),
  transports: [
    new winston.transports.Console()
  ]
});

logger.info('This is an informational message.');
```

Logging to file:

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

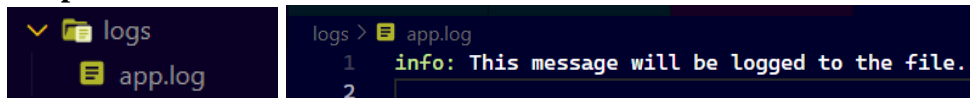
const logDir = 'logs';

// Create logs directory if not exists
if (!fs.existsSync(logDir)) {
  fs.mkdirSync(logDir);
}

const logger = winston.createLogger({
  level: 'info',
  format: winston.format.simple(),
  transports: [
    new winston.transports.File({ filename: `${logDir}/app.log` })
  ]
});

logger.info('This message will be logged to the file.');
```

Outputs:

The image shows two side-by-side screenshots. The left screenshot is a file explorer window showing a folder named 'logs' containing a file named 'app.log'. The right screenshot is a terminal window with the command 'logs > app.log' and its output: '1 info: This message will be logged to the file.' and '2'.

- **Morgan package:**

Morgan is a popular HTTP request logger middleware for Node.js applications. It provides a simple way to log incoming HTTP requests, including details such as request method, URL, status code, response time, and more. Morgan is often used with Express.js, a web application framework for Node.js.

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

const app = express();

// Use Morgan middleware with 'dev' format
app.use(morgan('dev'));

// Your Express routes and other middleware go here
app.get('/', (req, res) => {
  res.send('Welcome to Home Page!');
});

// Start the server
const PORT = process.env.PORT || 9000;
app.listen(PORT, () => {
  console.log(`Server is running on port: http://localhost:${PORT}`);
});
```

Output:

```
PS D:\cmdApps> node "d:\cmdApps\expLog.js"
Server is running on port: http://localhost:9000
GET / 304 7.736 ms - -
GET / 304 0.666 ms - -
GET / 304 0.634 ms - -
GET / 304 0.700 ms - -
GET / 304 0.665 ms - -
```

3. Keeping App Running:

- **PM2:** is a production process manager for Node.js applications with a built-in load balancer. It allows you to keep applications alive forever, to reload them without downtime and to facilitate common system admin tasks.

Installations: npm install pm2@latest -g

Start: pm2 start app_name

id	name	namespace	version	mode	pid	uptime	Ⓜ	status	cpu	mem	user	watching
1	app	default	1.0.0	fork	8052	9m	0	online	0%	42.9mb	Nis...	disabled

PS D:\cmdApps> pm2 start app.js

Managing processes:

Managing application state is simple here are the commands:

- pm2 restart app_name
- pm2 reload app_name
- pm2 stop app_name
- pm2 delete app_name

For more info: <https://pm2.keymetrics.io/docs/usage/quick-start/>

- **forever:** "Forever" is another process manager for Node.js, similar to PM2. It's used to ensure that a given script runs continuously by automatically restarting it if it crashes.

For more info: <https://www.npmjs.com/package/forever>

Installations: npm install forever -g

- **nohup:** nohup (short for "no hang up") is a command in Unix-like operating systems that is used to run another command in the background and detach it from the current session. It's often used to ensure that a process continues running even if the user logs out or the terminal is closed.

4. Threads:

In Node.js, child processes, clusters, and worker threads are different mechanisms for achieving parallelism or concurrency, each with its own use cases and characteristics.

- **child process:**

Child processes in Node.js allow you to spawn a new operating system process, which can run independently of the main process.

They are useful for executing CPU-intensive tasks in parallel or interacting with external programs.

The child_process module provides methods like 'spawn', 'exec', and 'fork' for creating child processes.

```
const { fork } = require('node:child_process');
const child = fork('child.js');

child.on('message', (message) => {
  console.log('Received message from child process: ${message}');
});
```

child.js:

```
// Simulate some asynchronous task in the child process
setTimeout(() => {
  // Send a message back to the parent process
  process.send('Hello from the child process!');
}, 1000);
```

Output:

```
PS D:\cmdApps> node .\childprocess.js
Received message from child process: Hello from the child process!
```

- **Cluster Module:**

The cluster module is designed to enable the distribution of incoming network traffic across multiple processes (workers).

It allows you to create a master process that manages a set of worker processes.

Each worker is a separate instance of the same Node.js application, sharing the same port.

```
const cluster = require('cluster');
const numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  cluster.on('exit', (worker, code, signal) => {
    console.log(`Worker ${worker.process.pid} died`);
  });
} else {
  // Worker code
  console.log(`Worker ${process.pid} started`);
}
```

```
PS D:\cmdApps> node clusterr.js
Worker 32484 started
Worker 9136 started
Worker 10092 started
Worker 15760 started
Worker 19624 started
```

- **Worker Threads:**

Worker threads were introduced to provide a way to run JavaScript code in parallel within a single Node.js process.

They are suitable for CPU-intensive tasks and can share memory using SharedArrayBuffer.

The worker_threads module is used to create and communicate with worker threads.

```
const { Worker } = require('worker_threads');

const worker = new Worker('worker.js', { workerData: {} });

worker.on('message', (message) => {
  console.log(`Received message from worker: ${message}`);
});
```

worker.js:

```
// Simulating some time-consuming task
Codeium: Refactor | Explain | X | Codiumate: Options | Test this function
const simulateTask = () => {
  for (let i = 0; i < 1000000000; i++) {
    // Performing some calculations
  }
};

// Performing the time-consuming task in the worker thread
simulateTask();

// Sending a message back to the main thread
const messageToMain = 'Task in the worker thread is complete!';
postMessage(messageToMain);
```