**UNIT-2**

**Introduction to Web programming in JavaScript**

Created in 1995 by Brandon Eich of Netscape/Mozilla originally called "Live Script" to match Netscape branding renamed to JavaScript to capitalize on popularity of Java submitted as a standard to ECMA in 1997 as ECMAScript" not directly related to Java Eich claims he was most influenced by Self and Scheme some JS syntax, libraries, etc. are ripped off by Java, C

**Creating Object**

This is a straightforward method where you directly define the object and its properties using curly braces {}:

let user = {

name: "John",

age: 30

};

console.log(user)

**result:** {name: 'John', age: 30}

**Access Object Properties**

You can access the **value** of a property by using its **key**.

1. **Using Dot Notation**

const dog = {

name: "Rocky",

};

console.log(dog.name);

**Output**: Rocky

1. **Using Bracket Notation**

const cat = {

name: "Luna",

};

console.log(cat["name"]);

**Output**: Luna

**Functions:**

A Function is a block of code that is designed to perform a task and executed when it is been called or invoked.

There are 3 ways of writing a function in JavaScript:

* + Function Declaration
  + Function Expression
  + Arrow Function

function ***name*** (***paramName***, ..., ***paramName***)

{

***statements***;

}

function myFunction(name) {

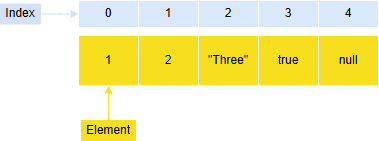
print ("Hello, " + name + “! \n");

print ("How are you? \n");

}

**Arrays:**

In JavaScript, an array is an ordered list of values. Each value is called an *element* specified by an *index*:



A JavaScript array has the following characteristics:

1. First, an array can hold values of mixed types. For example, you can have an array that stores elements with the types number, string, boolean, and null.
2. Second, the size of an array is dynamic and auto-growing. In other words, you don’t need to specify the array size up front.

const age = [17, 18, 15, 19, 14];

**Access Elements of an Array**

let numbers = [10, 30, 40, 60, 80]

// access first element

console.log(numbers[0]); // 10

// access third element

console.log(numbers[2]); // 40

**ES6 JavaScript**

JavaScript was invented by Brendan Eich in 1995, and became an ECMA standard in 1997.

* European Computer Manufacturers Association (ECMA)
* ECMAScript is the official name of the language.
* ECMAScript versions have been abbreviated to ES1, ES2, ES3, ES5, and ES6.
* Since 2016, versions are named by year (ECMAScript 2016, 2017, 2018, 2019, 2020).

1. **Let and const**

|  |  |  |
| --- | --- | --- |
| Var | Let | const |
| Global o | Block | Global or Block |
| var a = 10  function f() {  console.log(a)  }  f();  console.log(a); | let a = 10;  function f()  {  let b = 9  console.log(b);  console.log(a);  }  f(); | const a = 10;  function f() {  a = 9  console.log(a)  }  f(); |
| can re-declare the variable using | Users cannot re-declare the variable defined  but can update it. | users declare a const variable, they need to initialize it, otherwise, it returns an error |

**1. Scope Difference**

function testVar() {

if (true) {

var a = 10;

}

console.log(a);

}

function testLet() {

if (true) {

let b = 20;

}

console.log(b);

}

function testConst() {

if (true) {

const c = 30;

}

console.log(c);

}

testVar();

testLet();

testConst();

**2. Re-declaration**

// var

var x = 5;

var x = 10;

console.log(x); // 10

// let

let y = 5;

// let y = 10;

// const

const z = 5;

// const z = 10;

**3. Update (Re-assignment)**

// var

var a = 100;

a = 200;

console.log(a);

// let

let b = 300;

b = 400;

console.log(b);

// const

const c = 500;

c = 600;

console.log(c);

**2. Arrow Function**

Arrow functions provide you with an alternative way to write a shorter syntax compared to the function expression.

**Arrow function {()=>}**

* **Arrow functions** are anonymous functions i.e. functions without a name and are not bound by an identifier.
* Arrow functions do not return any value and can be declared without the function keyword.
* They are also called **Lambda Functions**.

|  |  |
| --- | --- |
| **Regular function** | **Arrow Function** |
| let x = function <function\_name>(parameters)  { // body of the function  }; | let x = (parameters) =>  { // body of the function }; |
| let add = function (x, y)  {  return x + y; };  console.log(add(10, 20)); // 30 | let add = (x, y) => x + y;  console.log(add(10, 20));  // 30; |

1. **Destructuring objects and Arrays:**

**Destructuring objects:**

**Before ES6:**

const person = {

  name: 'tim',

  age: 25,

  gender: 'male'

}

console.log(age);

**output:** c:\Users\rahulkarthik\Desktop\hello.js:6

console.log(age);

            ^

ReferenceError: age is not defined

    at Object.<anonymous> (c:\Users\rahulkarthik\Desktop\hello.js:6:13)

    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

Node.js v18.17.1

**With ES6:**

const person = {

  name: 'tim',

  age: 25,

  gender: 'male'

}

let { name, age, gender } = person;

console.log(name);

console.log(age);

console.log(gender);

**output:**

tim

25

Male

**Example 2: Renaming Variables**

const user = {

username: 'joe',

email: 'john@example.com'

};

const { username: uName, email: mail } = user;

console.log(uName);

console.log(mail);

**Destructuring Arrays:**

const arrValue = ['one', 'two', 'three'];

const [x, y, z] = arrValue;

console.log(x);

console.log(y);

console.log(z);

**output:**

one

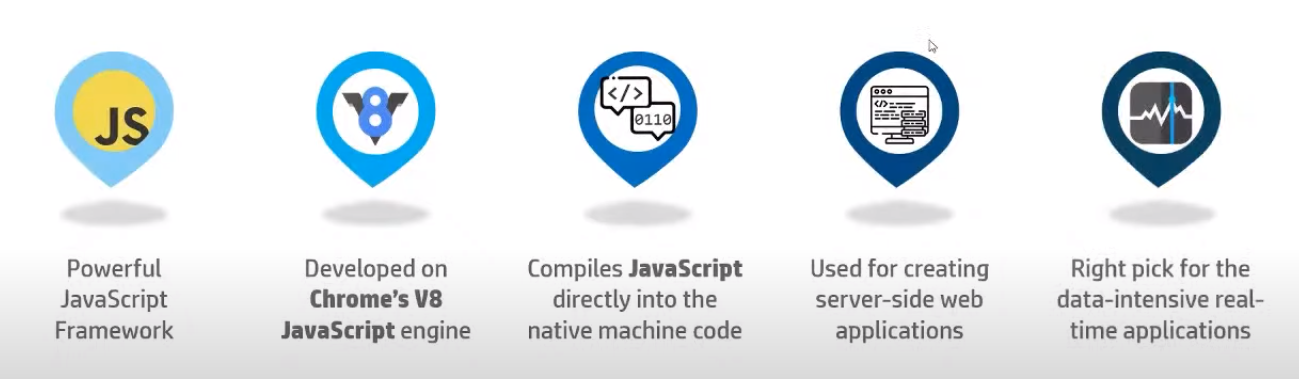
two

three

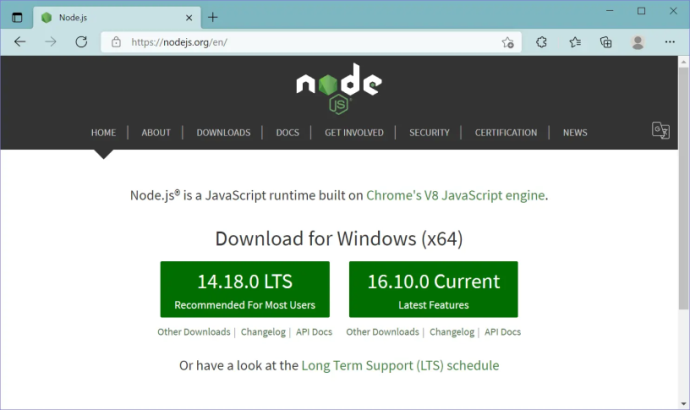
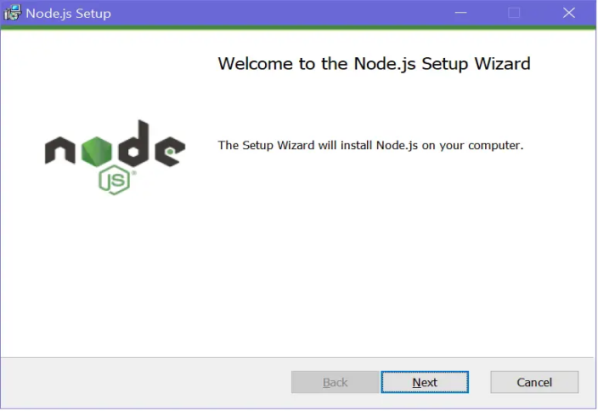


**Introduction to NodeJS**

* Node.js was developed and introduced by Ryan Dahl in 2009. Node.js
* can be used to create a variety of applications, including web apps, real-time chat apps, command-line apps and REST API servers.
* Mainly used to build network programs such as web servers.
* The standard filename extension of Node.js is .js.
* The engine that Node.js uses under the hood, V8 (Chrome V8)



**Installation of NodeJS**

**npm** (**Node Package Manager**) is the default package manager for Node.js, a JavaScript runtime environment. It plays a crucial role in the Node.js ecosystem by allowing developers to share and manage packages of code. Here’s a breakdown of what npm is and how it works:

1. Package Manager: npm is a command-line tool that helps you install, update, and manage dependencies (packages) in your Node.js projects.
2. Online Repository: It also includes an online database of public and private packages, known as the npm registry, where developers can publish and share their code.

**How to use npm (Node Package Manager)?**

* **Step 1**: Open Command Prompt

On Windows, search for "Command Prompt" in the Start menu and open it.

* **Step 2**: Check Node.js and npm Installation

Before using npm, ensure that Node.js and npm are installed. You can check this by running the following commands:

node -v

npm -v

If both commands return version numbers, you have Node.js and npm installed. If not, download and install Node.js from the [official website](https://nodejs.org/), which includes npm.

* **Step 3**: Create a New Directory for Your Project

Navigate to the location where you want to create your project and create a new directory:

mkdir my-node-app

cd my-node-app

* **Step 4**: Initialize a New Node.js Project

Run the following command to create a package.json file, which will hold metadata about your project:

npm init

You will be prompted to enter details about your project (name, version, description, etc.). You can press Enter to skip through the prompts or use npm init -y to accept the default values.

* **Step 5:** Install a Package

To install a package, use the following command:

npm install package-name

npm install -g nodemon

nodemon index.js

For example, to install the Express framework, run:

npm install express

This command will create a node\_modules directory and add the package to your package.json file under dependencies.

**Popular Node Packages**

Most popular Node.js packages that are widely used are:

1. Express:
   * A fast, unopinionated, minimalist web framework for Node.js, used for building web applications and APIs.
2. Mongoose:
   * An ODM (Object Data Modeling) library for MongoDB and Node.js, providing a schema-based solution to model application data.
3. Axios:
   * A promise-based HTTP client for the browser and Node.js, used for making HTTP requests to external APIs.
4. Socket.io:
   * A library for real-time web applications, enabling real-time, bidirectional communication between web clients and servers.
5. Body-Parser:

* Middleware to parse incoming request bodies in middleware before your handlers

1. Nodemon:

* A utility that will monitor for any changes in your source and automatically restart your server.

1. Dotenv:
   * A zero-dependency module that loads environment variables from a .env file into process.env, helping manage configuration settings.

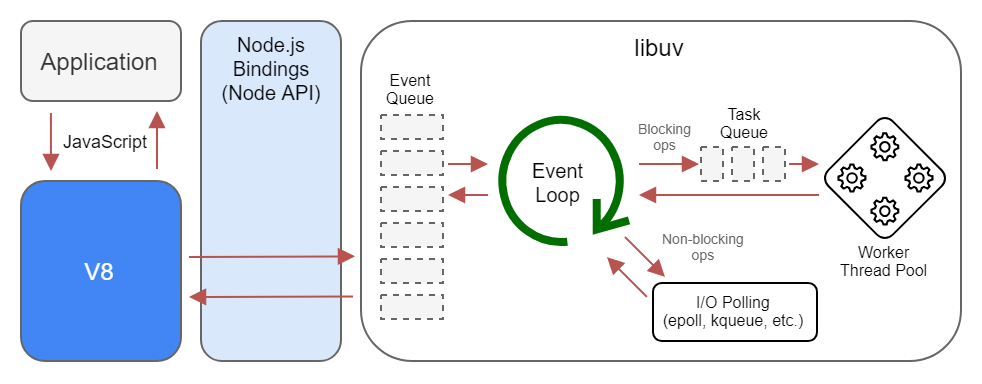
**Node Process Mechanism**

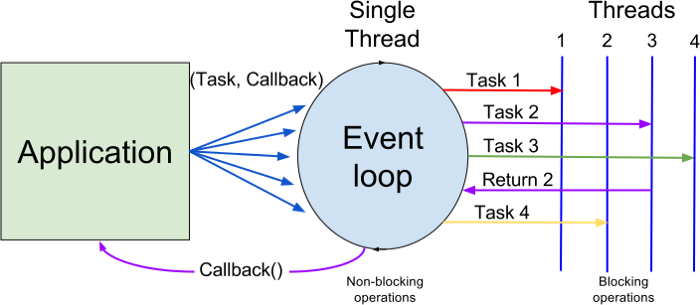
## Traditional Web Server Model

## 

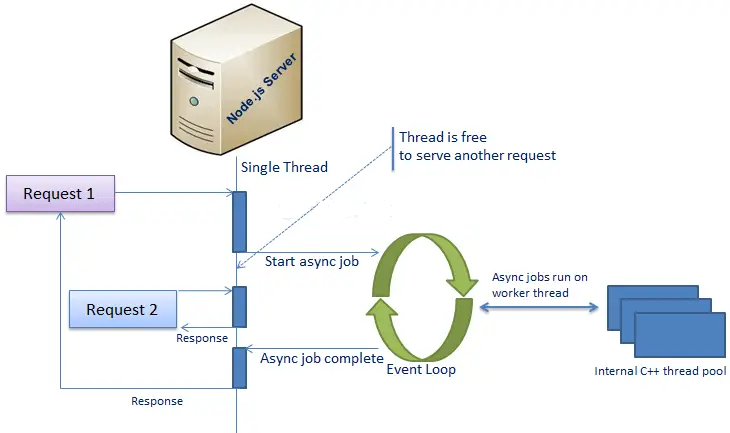
In the traditional web server model, each request is handled by a dedicated thread from the thread pool. If no thread is available in the thread pool at any point of time, then the request waits till the next available thread. Dedicated thread executes a particular request and does not return to thread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

* For a every client request, server will create a separate thread to handle the client request
* If a thread acquires a lock in the shared resources and it is exclusive lock, it will block other threads

****

****

Node.js processes user requests differently when compared to a traditional web server model. Node.js runs in a single process and the application code runs in a single thread and thereby needs less resources than other platforms. All the user requests to your web application will be handled by a single thread and all the I/O work or long running job is performed asynchronously for a particular request. So, this single thread doesn't have to wait for the request to complete and is free to handle the next request. When asynchronous I/O work completes then it processes the request further and sends the response. An event loop is constantly watching for the events to be raised for an asynchronous job and executing callback function when the job completes. Internally, Node.js uses libev for the event loop which in turn uses internal C++ thread pool to provide asynchronous I/O. The following figure illustrates asynchronous web server model using Node.js.



**Modules used in Node.js:**

1. fs: To handle the file system.
2. http: To create an HTTP server.
3. https: To create an HTTPS server.
4. path: To handle and transform file paths.
5. os: To get information about the operating system.

Modules allow you to break your code into reusable and organized pieces. You can export functions, objects, or values from one file and import them into another.

require():

* Used to import modules.
* Syntax: const moduleName = require('module').
* It loads modules synchronously at runtime.

module.exports:

* Used to export a module's functions, objects, or variables.
* Makes them available for other files to import using require().

How to Export and Import Modules:

**a)** Exporting from a Module (e.g., add.js):

* To make functions or objects available to other files, use module.exports.

add.js

function sumn(a, b) {

return a + b;

}

module.exports = { sumn }; // Exporting the function

**b)** Importing into a Module (e.g., main.js):

* Use require () to import the function or object from another module.

main.js

const { sumn } = require('./add'); // Importing the sumn function from add.js

console.log(sumn(10, 11)); // Logs 21 to the console

**Creating a HTTP server that supports Create, Read, operations via various HTTP methods**

**Process of creating application on Node Js**

A node.js web application contains the following three parts:

*Refer to below program*

* **Import the HTTP module:**

This line imports the built-in http module, which allows Node.js to transfer data over the HyperText Transfer Protocol (HTTP).

* **Create the server:**

This line creates an HTTP server that listens for requests and sends responses. The function passed to createServer is a callback that gets executed every time the server receives a request.

* **Set the response header**:

This line sets the HTTP status code to 200 (which means “OK”) and the content type to text/plain. This tells the client that the response will be plain text.

* **Send the response:**

This line sends the response body, which in this case is the string 'Hello World\n', and ends the response.

* **Listen on port 8081:**

This line makes the server listen for incoming requests on port 8081.

* **Log a message to the console:**

This line logs a message to the console indicating that the server is running and provides the URL where it can be accessed.

When we run this code, it starts a web server that listens on port 8081. If you navigate to http://127.0.0.1:8081/ in your web browser, you will see the message “Hello World”.

**Types of Modules in Node.js:**

* HTTP module

**Response -Plain text**

**Program: Hello.js**

var http = require("http");

http.createServer(function (request, response)

 {

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

   response.end('Hello World\n');

}).listen(8081);

console.log('Server running at http://127.0.0.1:8081/');

|  |  |
| --- | --- |
| **Running Program:**  Start> Node.js command Prompt  **Set path:**where you file **“ Hello.js”** available  **Cd:\User\Desktop\ node Hello.js**  **It shows the ‘**Server running at http://127.0.0.1:8081/');    Open any browser and type at url  127.0.0.1:8081 | Node.js first example 5 |

**Response-Html:**

const http=require("http");

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

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

res.write('<h1 style="color: Red">Hellooooo Allllll</h1>');

const a=

**`**<html>

    <head>

      <title>HTML File</title>

    </head>

    <body>

      <h1 style="color: Red">Hello All</h1>

      <h2 style="color: purple">HTTP module for web server creation</h2>

    </body>

  </html>

**`** ;

res.end(a);

});

server.listen(3000,()=>{

  console.log("Server running on http://localhost:3000");

})

(Or)

const http=require("http");

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

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

  res.write("<html>");

  res.write("<head><title>HTML File</title></head>");

  res.write("<body><h1>hellooooooooo</h1></body>");

  res.write("</html>");

  res.end();

});

server.listen(3000,()=>{

  console.log('Server running at http://localhost:3000/');

});

**Request**

const http = require('http');

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

const url = req.url;

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

    if (url === '/') {

    res.end('Hello! This is the Home Page.');

  } else if (url === '/about') {

    res.end('Welcome to the About Page!');

  } else if (url === '/contact') {

    res.end('This is the Contact Page.');

  } else {

    res.statusCode = 404;

    res.end('404 Page Not Found');

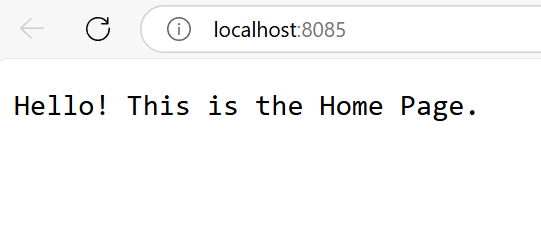
  }

}).listen(8085, () => {

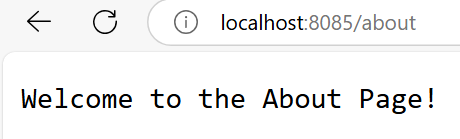
  console.log('Server running at http://localhost:8085/');

  });

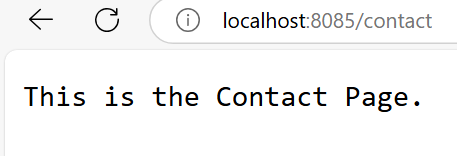
**Visiting http://localhost:8085/**

****

**Visiting http://localhost:8085/about**

****

**Visiting any other URL, such as http://localhost:8085/contact**

****

* **fs module**

fs stands for File System. It is a built-in module in Node.js that allows you to work with the file system on your computer. You can:

* Read files
* Write files
* Create files
* Delete files
* Rename files
* Watch file changes

**How to Import fs Module?**

const fs = require('fs');

**Synchronous and Asynchronous Methods in node.js:**

* Synchronous methods block the execution of code until the operation is complete.
* Asynchronous methods use callbacks, allowing non-blocking execution.

|  |  |
| --- | --- |
| **Synchronous (Sync)** | **Asynchronous (Async)** |
| Do one task at a time | Do many tasks at once |
| Waits for current task to finish | Doesn't wait — moves to next task |
| Slower for big tasks | Faster for big tasks |
| Blocks the rest of the code | Doesn’t block the code |

|  |  |
| --- | --- |
| **Sync Code**  const fs = require('fs');  const data = fs.readFileSync('ex.txt', 'utf8');  console.log(data);  console.log('Done');  **OUTPUT:**  hellllllllllllllllllloooooooooooooooo  Done | **Async Code:**  const fs = require('fs');  fs.readFile('ex.txt', 'utf8', (err, data) => {  console.log(data);  });  console.log('Done');  **OUTPUT:**  Done  hellllllllllllllllllloooooooooooooooo |

**Create a File:**

To create a new file and write content to it, you can use the fs.writeFile function.

const fs = require('fs');

fs.writeFile('example.txt', 'Hello, Node.js!', (err) => {

if (err) throw err;

console.log('File created successfully’);

});

**(or)**

const fs = require('fs');

const content = 'This is the content of the file.\n';

fs.writeFile('example.txt', content, 'utf8', (err) => {

if (err) {

console.error('Error creating the file:', err);

} else {

console.log ('File created successfully.');

}

});

**Read a File:**

To read the content of an existing file, you can use the fs.readFile function. Here's an

const fs = require('fs');

fs.readFile('example.txt', 'utf8', (err, data) => {

if (err) {

console.error('Error reading the file:', err);

} else {

console.log('File content:\n', data);

}

});

**Append**

const fs = require('fs');

fs.appendFile('ex1.txt', '\nNew line added!', (err) => {

  if (err) throw err;

  console.log('Content appended');

});

**delete**

To delete a file in Node.js, you can use the fs.unlink function from the fs module.

const fs = require('fs');

fs.unlink('ex.txt', (err) => {

  if (err) throw err;

  console.log('File deleted');

});

**(or)**

const fs = require('fs');

const filePath = 'example.txt';

fs.unlink(filePath, (err) => {

if (err) {

console.error('Error deleting the file:', err);

} else {

console.log('File deleted successfully.');

}

});

**fs and http -Create a file with file extension(.html)**

**Serving HTML via Node.js**

If you want to serve the HTML content over HTTP using Node.js, you can use the http module to create a simple server that serves the HTML file. Here’s an example:

const http = require('http');

const fs = require('fs');

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

  fs.readFile('trans.html', 'utf8', (err, data) => {

    if (err) {

      res.writeHead(500, { 'Content-Type': 'text/plain' });

      res.end('Error loading the page');

      return;

    }

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

    res.end(data);

  });

}). listen (8085, () => {

  console.log ('Server running at http://localhost:8085/');

});

**trans.html**

<html>

<head>

<style>

img {

  background: cyan;

  border-radius: 3% 90%;

  border: solid red;

  height: 400px;

  margin: 300px;

  transition: all 7s;

  transition-timing-function: ease-in-out;

  width: 300px;

}

img:hover {

  transform:  rotate(50deg);

}

</style>

</head>

<body>

<img src="https://tse1.mm.bing.net/th/id/OIP.VIXX4Hfvqze8Hi2xb1noxgHaJh?rs=1&pid=ImgDetMain&o=7&rm=3"></img>

</body>

</html>

* **Path module**

The path module is a built-in core module in Node.js that provides utilities for working with file and directory paths. It simplifies handling paths by:

* Managing differences in path separators (e.g., / on Linux/Mac vs \ on Windows).
* Supporting absolute and relative paths consistently.
* Providing methods to join, resolve, parse, and format paths in a platform-independent way.

## **Importing the Path Module**

const path = require('path');

**Methods:**

* path.resolve()
* path.join()
* path.basename()
* path.extname()
* path.dirname()
* path.isAbsolute()
* path.relative()

**Example:**

const fs = require('fs');

const path = require('path');

const filePath = path.join('C:\\Users\\rahulkarthik\\jim\\file.txt');

console.log('File Name:', path.basename(filePath));

console.log('Directory:', path.dirname(filePath));

console.log('Extension:', path.extname(filePath));

console.log('Is Absolute Path?', path.isAbsolute(filePath));

fs.readFile(filePath, 'utf8', (err, data) => {

    if (err) {

        console.error('Error reading file:', err);

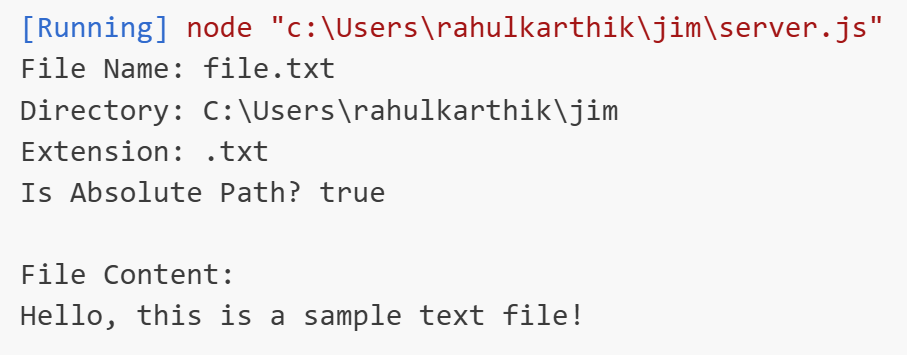
        return;

    }

    console.log('\nFile Content:');

    console.log(data);

});

****

* **OS module**

The os module in Node.js provides information about the operating system and utilities for interacting with it. It is a core module, so no installation is required.

**Importing the OS Module:** const os = require('os');

**Methods:**

The important methods of the os module in Node.js:

* os.arch()
* os.platform()
* os.type()
* os.release()
* os.totalmem()
* os.freemem()
* os.cpus()
* os.networkInterfaces()
* os.homedir()
* os.tmpdir()
* os.hostname()
* os.userInfo()

**Example:**

const os = require('os');

console.log('OS Type:', os.type());

console.log('Platform:', os.platform());

console.log('Architecture:', os.arch());

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

console.log('Hostname:', os.hostname());

console.log('Home Directory:', os.homedir());

console.log('Total Memory:', os.totalmem());

console.log('Free Memory:', os.freemem());

console.log('CPU Info:', os.cpus());

console.log(os.userInfo());

console.log(os.networkInterfaces());

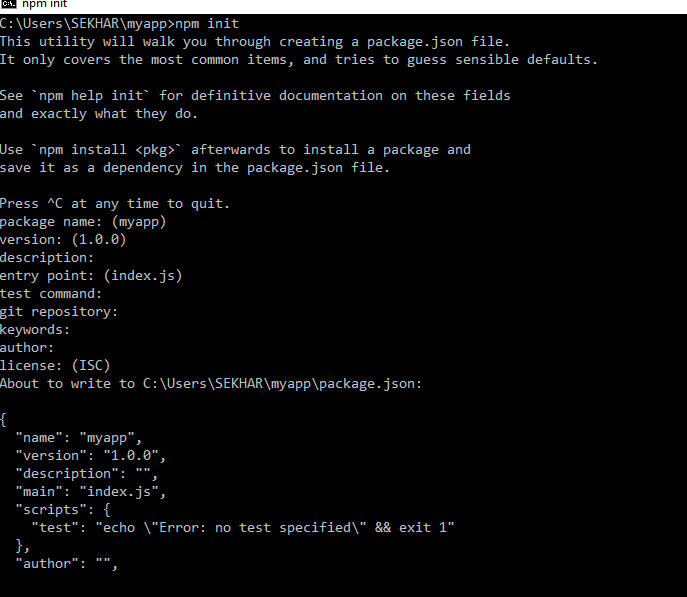
**Express.js**

Express provides a minimal interface to build applications. It provides us the tools that are required to build app. It is flexible as there are numerous modules available on npm, which can be directly plugged into Express.

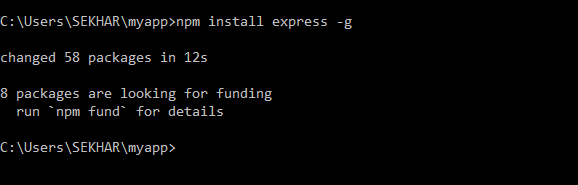
**Node Package Manager(npm)**

* npm allows us to access all these packages and install them locally.
* Whenever we create a project using npm, we need to provide a **package.json** file,
* Which has all the details about our project.

|  |  |
| --- | --- |
| Step- | Step-2  Now to create the package.json file using npm, use the following code |



**Step 3** − Now we have our package.json file set up, we will further install Express. To install Express and add it to our package.json file



**hello\_test.js**

var express = require('express');

var app = express();

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

{

res.send(" Welcome to Express- Thank you!");

});

app.listen(3000);

open your browser and go to **http://localhost:3000**

**Middleware**

**Middleware** acts like a bridge connecting requests and responses. Whether it’s for logging, user authentication, data parsing, or error handling — middleware functions play a pivotal role in every Express.js app.

When a user visits your website or makes an API request, the request goes through multiple steps before it gets a response. Middleware is one of those steps. It can read the request, make changes to it, add extra data, stop it, or pass it to the next step.

**Ex:** Think of it like a security checkpoint in an airport. Every person (request) must go through it. The staff (middleware) can check documents, add labels (data), or stop someone (return an error). If everything is okay, the person (request) moves to the next checkpoint (route).

**How Middleware Works?**

Let’s understand the flow of middleware step-by-step:

1. User sends a request (e.g., opens your site or hits an API)
2. The request enters your Express server
3. The request goes through one or more middleware functions (in order)
4. Each middleware can:
   * Do something with the request (like log it or validate)
   * Stop and return a response (like 403 Unauthorized)
   * Call next () to pass it along
5. If everything is okay, it finally reaches the route handler and returns a response

This system gives you full control over every request that enters your app.

**Syntax of a Middleware:**

var app = express();  
app.use(function (req, res, next) {  
 // body of middleware function.  
 next();  
})

* **(req, res, next) => {}:**This is the middleware function where you can perform actions on the request and response objects before the final handler is executed.
* **next():**This function is called to pass control to the next middleware in the stack if the current one doesn’t end the request-response cycle.

An Express application can use the following types of middleware:

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

1. **Application-level middleware**

Bind application-level middleware to an instance of the app object by using the app.use() and app.METHOD() functions, where METHOD is the HTTP method of the request that the middleware function handles (such as GET, PUT, or POST) in lowercase.

**Example:**

const express = require('express');

const app = express();

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

    if(10%2==1){

        console.log('Server');

        next();

    }

    else{

        res.send('Go Go');

    }

});

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

    res.send('Hello, Express!');

});

app.listen(3000, () => {

    console.log('Server running at http://localhost:3000');

});

1. **Router-level middleware**

Router-level middleware is applied to a specific router instance using router.use() or router.METHOD().

**Example:**

const express = require('express');

const app = express();

const router=express.Router();

router.use((req,res,next)=>{

    if(10<20){

        next();

    }

    else{

        res.send("hellooooooooooo");

    }

});

router.get('/hello',(req,res)=>{

    res.send("haiiii Hellooooooo");

});

app.use('/', router);

app.listen(3000, () => {

    console.log("Server running at http://localhost:3000");

});

1. **Error-handling middleware**

Error-handling middleware is a special type of middleware used to catch and respond to errors during the request-response cycle. It is defined with four parameters: err, req, res, next. This middleware is essential for sending a consistent error response and avoiding unhandled exceptions that might crash the server.

const express = require('express');

const app = express();

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

    next("Something went wrong!");

});

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

    res.status(500).send("Error: " + err);

});

app.listen(3000, () => {

    console.log("Server running at http://localhost:3000");

});

1. **Built-in middleware**

Express provides built-in middleware to help with common tasks, like serving static files or parsing data. For example, express.static() serves files like images, and express.json() helps parse incoming JSON data, express.urlencoded() -parses form data

1. **express.json()**

Parses incoming requests with JSON payloads.

Makes parsed JSON data available in req.body.

Used when clients send JSON data in requests, often in APIs.

**Example**:

const express = require('express');

const app = express ();

app.use(express.json());

app.post('/data', (req, res) => {

    console.log(req.body);

    res.send('Data received');

});

app.listen(3000, () => {

    console.log("Server running at http://localhost:3000");

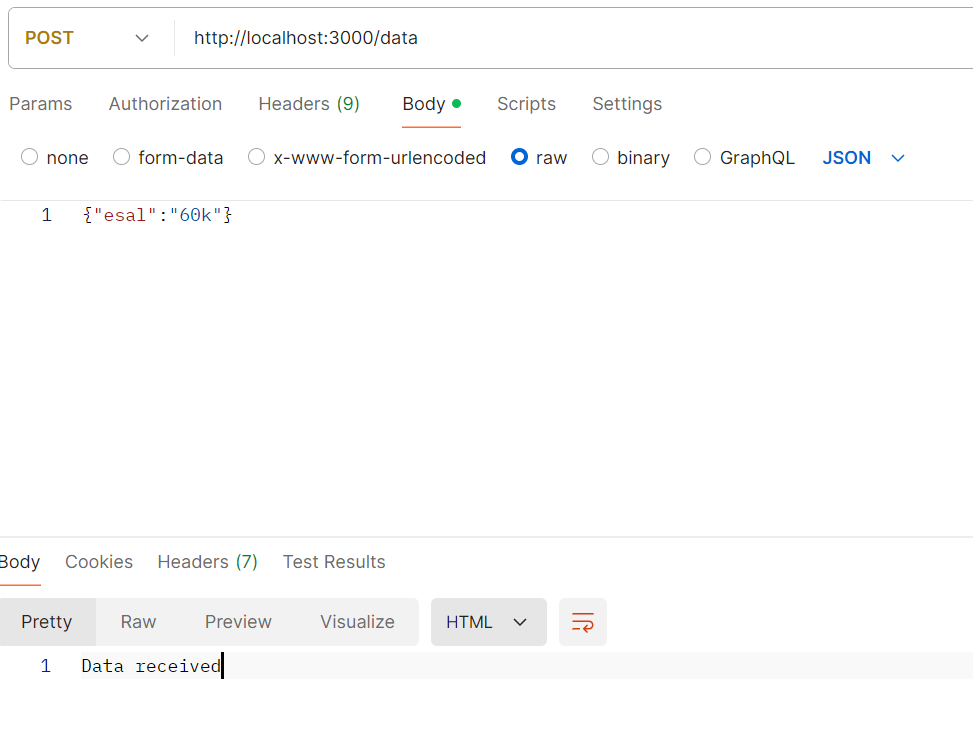
});

**Result:**

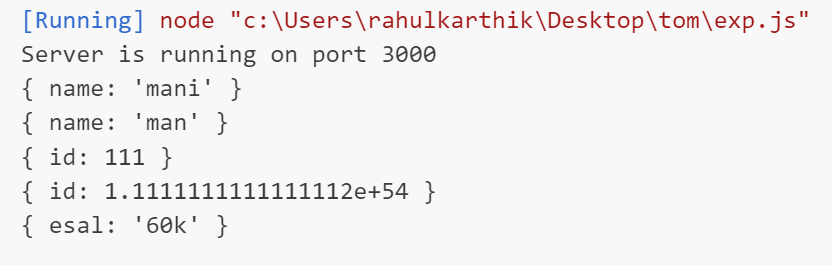
**In postman App, we can give as follows:**

If a POST request is made to /data with a JSON body like {"key": "value"},

And responds with: Data received



Now in console, we get



1. **express.static()**

It serves static files (HTML, CSS, images, JS) from a folder so the browser can load them directly without creating routes for each file.

**Server.js**

const express = require('express');

const app = express();

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

app.listen(3000, () => {

    console.log("Server running at http://localhost:3000");

});

Create a folder named **public.** Inside public, create a file index.html

**index.html**

<html>

<head>

    <title>Static Example</title>

</head>

<body>

    <h1>Lovely Krishna</h1>

    <img src="https://th.bing.com/th/id/OIP.TwFQvVmKf9ufQY7hQb-zoQHaHa?w=200&h=200&c=7&r=0&o=5&dpr=1.7&pid=1.7" alt="Image">

</body>

</html>

1. **express.urlencoded()**

Parses incoming requests with URL-encoded payloads (typically from HTML forms). Makes parsed form data available in req.body. Used when clients submit form data.

const express = require('express');

const app = express();

app.use(express.urlencoded({ extended: true }));

app.post('/submit', (req, res) => {

    console.log(req.body);

    res.send('Form data received');

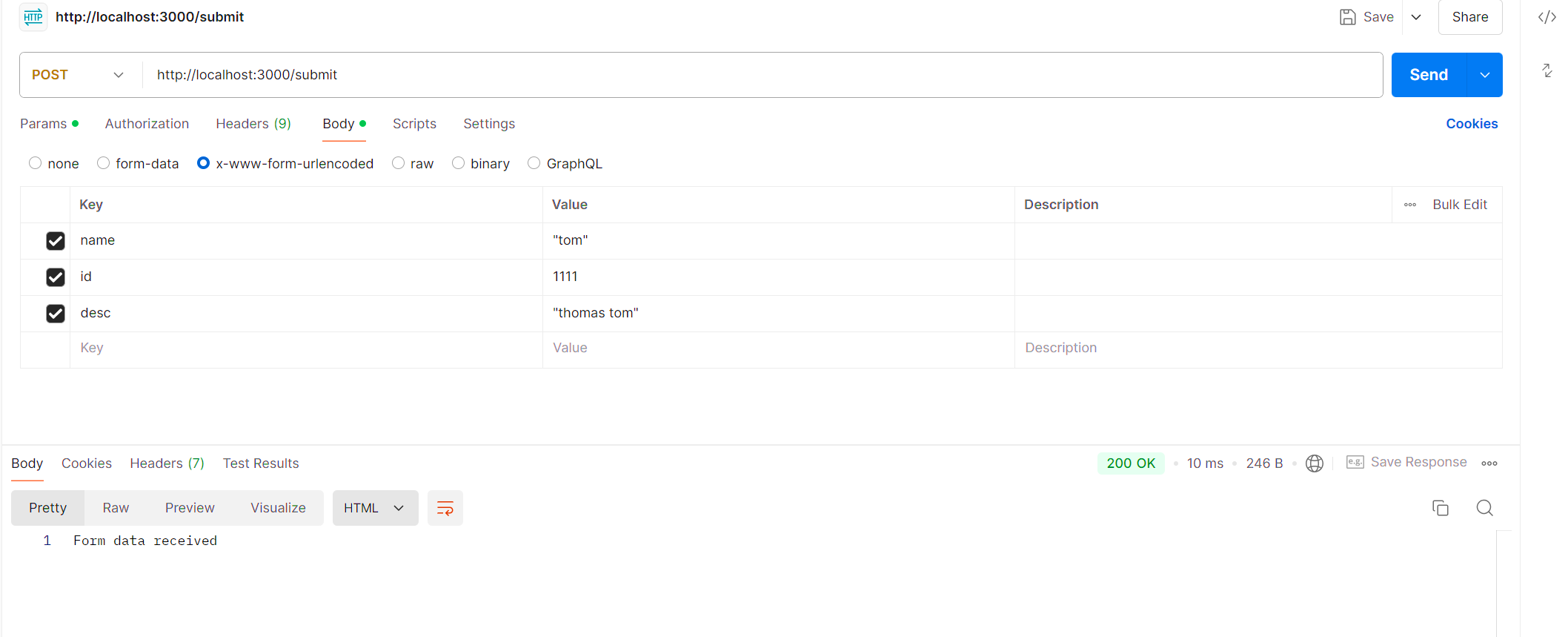
});

app.listen(3000, () => {

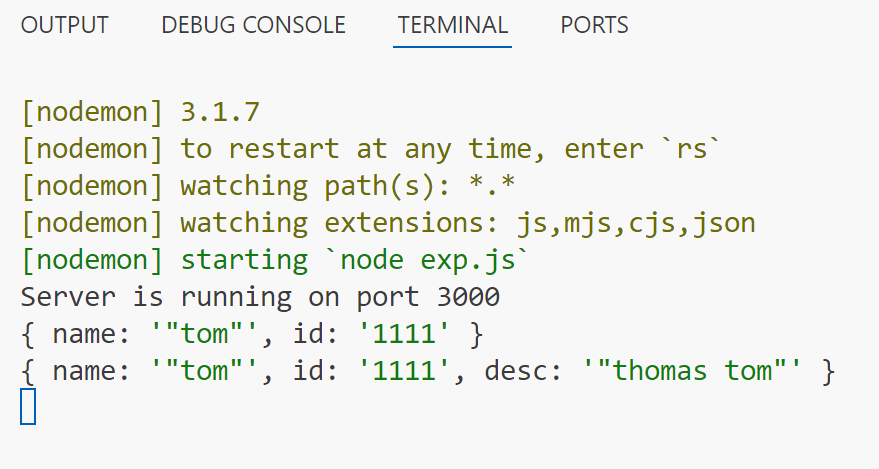
    console.log("Server running at http://localhost:3000");

});

**In Postman App,** set the Content-Type header to application/x-www-form-urlencoded and provide the form data in the correct format key:value

****

**Now, observe console, which is displayed as follows:**

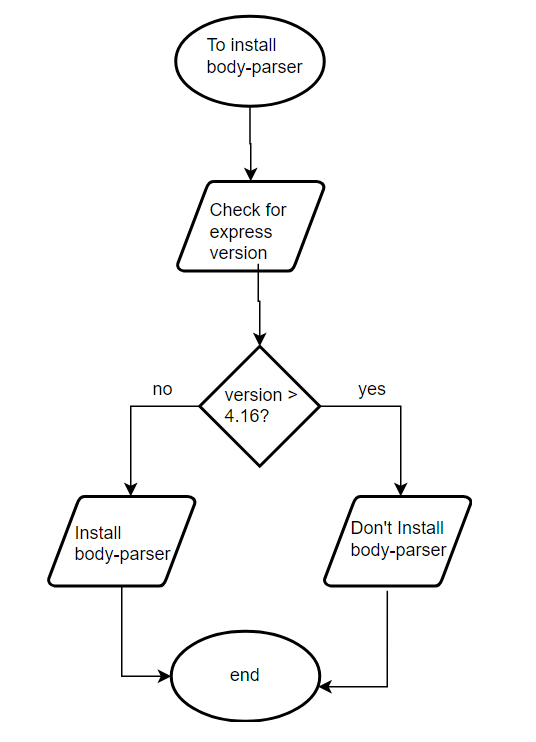
****

1. **Third party middleware**

Third-party middleware is developed by external developers and packaged as npm modules. These middleware packages add additional functionality to your application, such as request logging, security features, or data validation. For example, the Morgan middleware logs HTTP requests, body-parser helps parse incoming request bodies for easier handling of form data, cors enables cross-origin requests

**Body Parser:**

* Body-parser is a third-party middleware for processing incoming HTTP requests in Node.js. It is used to extract the request body and expose it on the request object for easier handling.
* When a client makes an HTTP request to a server, the request may contain data in the request body, such as form data or JSON data. Body-parser helps to extract this data and make it available in a format that can be easily used by JavaScript code running on the server.
* Body-parser supports various types of request bodies including JSON, URL-encoded data, raw text, and even multi-part data. It parses the request body and populates the request object with a body object containing the parsed data.
* Body-parser is an important tool for building web applications with Node.js because it simplifies the process of handling incoming data from client requests.
* **Installation of body-parser module:**



Latest 4.18.2

**npm install body-parser**

const express = require('express');

const bodyParser = require('body-parser');

const app = express();

app.use(bodyParser.json());

app.use(bodyParser.urlencoded({ extended: true }));

app.post('/submit', (req, res) => {

    console.log(req.body);

    res.send('Data received');

});

const PORT = 3000;

app.listen(PORT, () => {

    console.log(`Server is running on port ${PORT}`);

});

**REST API: CR operations**

const express = require("express");

const app = express();

app.use(express.json());

let data1=[];

app.post('/data',(req,res)=>{

  const ndata=req.body;

  data1.push(ndata);

  console.log("Data received:", data1);

  res.send("Data has been posted successfully.");

});

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

    res.json(data1);

});

app.listen(3000, () => {

    console.log("Server running at http://localhost:3000");

});

**REST API: CRUD operations**

const express = require("express");

const app = express();

app.use(express.json());

let data1 = [];

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

  data1.push(req.body);

  res.send("Posted");

});

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

  res.json(data1);

});

app.get("/data/:index", (req, res) => {

    const index = req.params.index;

    if (data1[index]) {

        res.json(data1[index]);

    } else {

        res.status(404).send("Data not found.");

    }

});

app.put("/data/:id", (req, res) => {

  let id = req.params.id;

  data1[id] = req.body;

  res.send("Updated");

});

app.delete("/data/:index", (req, res) => {

  const index = req.params.index;

  if (data1[index]) {

    data1[index] = null;   // instead of removing, we set null

    res.send("Data has been deleted successfully.");

  } else {

    res.status(404).send("Data not found.");

  }

});

app.listen(3000, () => {

  console.log("Server running at http://localhost:3000");

});

**Output:**

* 1. Start server
  2. Server runs at <http://localhost:3000/>
  3. **POST**: <http://localhost:3000/data>

Body → raw → JSON:

{"name":"Apple"}

* 1. Response: Posted
  2. Add another: {"name":"Mango"}
  3. Response: Posted
  4. **GET:** http://localhost:3000/
  5. Response:

[

{"name":"Apple"},

{"name":"Mango"}

]

* 1. **PUT**: http://localhost:3000/data/0

Body → raw → JSON:

{"name":"Orange"}

* 1. Response: Updated
  2. Now **GET** again → http://localhost:3000/

[

{"name":"Orange"},

{"name":"Mango"}

]

* 1. **DELETE**: http://localhost:3000/data/1
  2. Response: Deleted
  3. Now **GET** again → http://localhost:3000/

[

{"name":"Orange"}

]

**Template Engine**

A template engine is a tool that helps you combine static HTML with dynamic data.

Instead of writing plain HTML, you can write templates (HTML + placeholders).  
When the server runs, the template engine fills in the placeholders with actual data and produces the final HTML.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **ejs** | **mustache** | **nunjucks** | **pug** |
| **Description** | Embedded JavaScript templates | Logic-less templates with JavaScript | A rich and powerful templating language for JavaScript by **Mozilla** | A clean, whitespace-sensitive template language for writing HTML |
| **Licenses** | Apache | MIT | BSD-2-Clause | MIT |
| **Created** | Feb 2011 | Jan 2012 | Aug 2012 | Aug 2013 |
| **CLI Shortcut** | npm i ejs | npm i mustache | npm i nunjucks | npm i p |

**Embedded JavaScript (EJS) Templates:**

EJS stands for *Embedded JavaScript* is a Templating Language. EJS is very simple, light and fast. It allows us to create HTML Markup with Plain JavaScript.

EJS was founded in Feb 2011 and is licensed under Apache License, version 2.0. EJS is the Top most downloaded template engine on npm with approx 6,115,876 downloads per week.

If you want to use Embedded JavaScript (EJS) as a template engine in your Express app so you can render HTML dynamically instead of just sending JSON or plain text.

* 1. **Install EJS**

npm install ejs

* 1. **app.js:**

const express = require("express");

const app = express();

app.use(express.json());

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

let dataStorage = [];

app.post('/data', (req, res) => {

    const newData = req.body;

    dataStorage.push(newData);

    console.log("Data received:", newData);

    res.send("Data has been posted successfully.");

});

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

    res.render("data.ejs", { items: dataStorage });

});

app.listen(3000, () => {

  console.log("Server running at http://localhost:3000");

});

* 1. **Create the EJS Template**

Inside your project folder, create a views directory (EJS looks here by default).

**views/data.ejs**

<html>

<head>

    <title>Stored Data</title>

</head>

<body>

    <h1>EJS Templates</h1>

    <ul>

        <% items.forEach(function(item) { %>

            <li>

                Name: <%= item.name %>, Age: <%= item.age %>

            </li>

        <% }); %>

    </ul>

</body>

</html>

**Add Data in index.ejs**

To bind data in index.ejs file, we can use <%= %> template literals. Inside template literals, i.e. <%= %>, we can pass any datatype.

index.ejs:

<h2>Name is <%= name %> and id is <%= id %> </h2>

server.js:

res.render('index',{name:"avinash",id:212});

**If condition**

If Condition in EJS is used to test variable, and then print its property.

index.ejs:

<% if (user) { %>

<p><%= user.name %></h2>

<p><%= user.id %></h2>

<% } %>

server.js:

res.render('index',{user:{name:'avinash',id:212}});

**forEach**

forEach loop is used to traversal all data inside an array list.

index.ejs:

<div class="data">

<h2>Data from Array</h2>

<ol>

<% month.forEach(function(name) { %>

<li><%= name %></li>

<% }); %>

</ol>

</div>

server.js:

res.render('index',{month:['jan','feb','mar']});

**If–Else Example**

<% if (user.age >= 18) { %>

<p><%= user.name %> is an Adult</p>

<% } else { %>

<p><%= user.name %> is a Minor</p>

<% } %>

**ForEach Loop Example**

<ul>

<% students.forEach(student => { %>

<li><%= student.name %> - <%= student.age %></li>

<% }) %>

</ul>

**For Loop Example**

<ul>

<% for (let i = 0; i < students.length; i++) { %>

<li><%= students[i].name %></li>

<% } %>

</ul>

**Without Template Engine**

plain HTML (index.html):

<h1>Welcome Rahul</h1>

<p>You are 22 years old.</p>

If the user changes, we have to manually change the file each time.

**With Template Engine (EJS)**

Template (index.ejs):

<h1>Welcome <%= user.name %></h1>

<p>You are <%= user.age %> years old.</p>

Data from server (app.js):

res. render ('index', { user: { name: "Rahul", age: 22 } });

Final Rendered HTML in Browser:

<h1>Welcome Rahul</h1>

<p>You are 22 years old.</p>

Now, if user.name = "Meena", the same template works and displays:

<h1>Welcome Meena</h1>

<p>You are 19 years old.</p>