**UNIT – 3 Node.js and Express.js**

**Node.js:** It is an open-source, cross-platform JavaScript runtime environment that executes JavaScript code outside of a web browser. It uses the V8 JavaScript engine, which is developed by Google and powers Google Chrome. Node.js allows developers to run JavaScript on the server-side, enabling them to build scalable and high-performance network applications.

**Overview of Node.js**:

Asynchronous and Event-Driven: One of the key features of Node.js is its asynchronous, event-driven architecture. Instead of blocking I/O operations, Node.js uses non-blocking, asynchronous I/O, allowing it to handle multiple concurrent connections efficiently. This makes Node.js particularly suitable for building real-time applications like chat applications or streaming services.

Single-Threaded, Non-Blocking: Node.js operates on a single-threaded event loop model. This means that it can handle many connections simultaneously without the need for threading. When Node.js performs an I/O operation, it does not wait for it to complete. Instead, it continues to execute other code, and when the I/O operation is finished, it triggers a callback. This non-blocking nature contributes to its high performance and scalability.

NPM (Node Package Manager): Node.js comes with npm, a package manager that hosts thousands of reusable libraries and modules. npm simplifies the process of adding functionality to Node.js applications by providing a vast ecosystem of pre-built modules that can be easily integrated into projects.

Cross-Platform: Node.js is cross-platform, meaning it runs on various operating systems such as Windows, macOS, and Linux. This allows developers to write applications using Node.js that can run on different environments without modification.

Significance in Backend Development: Efficiency and Performance: Node.js enables developers to build highly efficient and performant backend systems. Its non-blocking, event-driven architecture allows for handling a large number of concurrent connections with minimal resources, making it suitable for real-time applications and microservices.

Full-Stack JavaScript Development: With Node.js, developers can use JavaScript for both frontend and backend development. This allows for easier code sharing, better collaboration between frontend and backend developers, and faster development cycles.

Scalability: Node.js is designed for scalability, allowing applications to easily scale horizontally by adding more nodes to distribute the workload. Its lightweight architecture and support for asynchronous I/O make it well-suited for handling high traffic and large scale applications.

Community and Ecosystem: Node.js has a vibrant and active community, with a vast ecosystem of libraries, frameworks, and tools. This ecosystem provides developers with a wide range of options for building and maintaining backend systems, from web frameworks like Express.js to database connectors like MongoDB driver.

**Node JS Set-up**

Setting up Node.js involves installing Node.js and npm (Node Package Manager) and configuring your environment. Follow these steps:

**Step 1: Download and Install Node.js**

1. Visit the official Node.js website.
2. Download the **LTS (Long-Term Support)** version for stability.
3. Run the installer and follow the installation steps:
   * Accept the license agreement.
   * Choose the installation path.
   * Ensure **"Install npm"** is checked.
   * Complete the installation.

**Step 2: Verify Installation**

After installation, open **Command Prompt (Windows)** or **Terminal (Mac/Linux)** and run:

sh

CopyEdit

node -v

This should display the installed Node.js version.

Similarly, check npm with:

sh

CopyEdit

npm -v

**Step 3: Install a Package Manager (Optional)**

* **npm** is included by default.
* Alternatively, install **Yarn** (another package manager):

sh

CopyEdit

npm install -g yarn

**Step 4: Set Up a Node.js Project**

1. Create a project folder:

sh

CopyEdit

mkdir my-node-app && cd my-node-app

1. Initialize a Node.js project:

sh

CopyEdit

npm init -y

This creates a package.json file.

**Step 5: Install Dependencies**

To install packages, use:

sh

CopyEdit

npm install <package-name>

Example:

sh

CopyEdit

npm install express

**Step 6: Create and Run a Simple Node.js Script**

1. Create index.js:

sh

CopyEdit

touch index.js # (Linux/macOS)

echo > index.js # (Windows)

1. Add a simple script:

js

CopyEdit

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

1. Run the script:

sh

CopyEdit

node index.js

**Step 7: Use a Local Server (Optional)**

If using Express.js:

1. Install Express:

sh

CopyEdit

npm install express

1. Create server.js and add:

js

CopyEdit

const express = require("express");

const app = express();

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

res.send("Hello, World!");

});

app.listen(3000, () => {

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

});

1. Run the server:

sh

CopyEdit

node server.js

1. Open http://localhost:3000 in your browser.

**Node.js Basics**

Node.js is a **JavaScript runtime** built on Chrome's **V8 engine**, allowing you to run JavaScript code **outside the browser**. It is widely used for building **server-side applications, APIs, and microservices**.

**1. Features of Node.js**

✅ **Asynchronous & Non-blocking** – Handles multiple requests without waiting for previous ones to complete.  
✅ **Single-threaded but Highly Scalable** – Uses event-driven architecture to manage concurrency.  
✅**Cross-platform** – Runs on Windows, macOS, and Linux.  
✅ **Uses JavaScript** – Makes it easy for front-end developers to transition to back-end development.  
✅ **Built-in npm (Node Package Manager)** – Allows installing third-party packages easily.

**2. Installing Node.js**

1. Download from Node.js Official Website.
2. Install the **LTS version** for stability.
3. Verify installation:

sh

CopyEdit

node -v

npm -v

**3. Running a Simple Node.js Script**

1. Create a new file app.js
2. Add the following code:

js

CopyEdit

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

1. Run the file:

sh

CopyEdit

node app.js

**Output:** Hello, Node.js!

**4. Node.js Modules**

Node.js has built-in modules like fs, http, and path. You can also create custom modules.

**4.1 Built-in Modules Example (http)**

Create an HTTP server using the http module:

js

CopyEdit

const http = require("http");

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

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

res.end("Hello, World!");

});

server.listen(3000, () => {

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

});

Run it and visit http://localhost:3000 in your browser.

**5. npm (Node Package Manager)**

**5.1 Initialize a Node.js Project**

sh

CopyEdit

npm init -y

This creates a package.json file.

**5.2 Installing Packages**

Install express (a web framework):

sh

CopyEdit

npm install express

**6. Creating a Simple Express Server**

js

CopyEdit

const express = require("express");

const app = express();

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

res.send("Welcome to my Node.js app!");

});

app.listen(3000, () => {

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

});

Run it using:

sh

CopyEdit

node app.js

**7. Handling Asynchronous Code**

Node.js uses callbacks, promises, and async/await to handle asynchronous operations.

**7.1 Callback Example**

js

CopyEdit

const fs = require("fs");

fs.readFile("file.txt", "utf8", (err, data) => {

if (err) throw err;

console.log(data);

});

**7.2 Using Promises**

js

CopyEdit

const fs = require("fs").promises;

fs.readFile("file.txt", "utf8")

.then(data => console.log(data))

.catch(err => console.error(err));

**7.3 Using async/await**

js

CopyEdit

const fs = require("fs").promises;

async function readFile() {

try {

const data = await fs.readFile("file.txt", "utf8");

console.log(data);

} catch (err) {

console.error(err);

}

}

readFile();

**8. Node.js File System (fs)**

Read a file:

js

CopyEdit

const fs = require("fs");

fs.readFile("file.txt", "utf8", (err, data) => {

if (err) throw err;

console.log(data);

});

Write to a file:

js

CopyEdit

fs.writeFile("file.txt", "Hello, Node.js!", (err) => {

if (err) throw err;

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

});

**9. Exporting and Importing Modules**

**9.1 Creating a Module (math.js)**

js

CopyEdit

exports.add = (a, b) => a + b;

exports.sub = (a, b) => a - b;

**9.2 Importing in Another File (app.js)**

js

CopyEdit

const math = require("./math");

console.log(math.add(5, 3)); // Output: 8

console.log(math.sub(5, 3)); // Output: 2

**10. Handling HTTP Requests with Express**

js

CopyEdit

const express = require("express");

const app = express();

app.use(express.json());

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

res.send(`Received: ${JSON.stringify(req.body)}`);

});

app.listen(3000, () => {

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

});

**Conclusion**

These are the core basics of Node.js. From here, you can explore:

* **Databases** (MongoDB, MySQL)
* **Authentication** (JWT, OAuth)
* **WebSockets** (real-time apps)
* **Microservices** and more!

**Node.js Console**

The **console** module in Node.js is used for debugging and logging information to the terminal. It provides methods like console.log(), console.error(), console.warn(), and more.

**1. Basic Console Methods**

**1.1 console.log() - Standard Output**

js

CopyEdit

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

**Output:**

CopyEdit

Hello, Node.js!

**1.2 console.error() - Error Logging**

js

CopyEdit

console.error("This is an error message!");

**Output:**

vbnet

CopyEdit

This is an error message!

*(Usually appears in red in the terminal.)*

**1.3 console.warn() - Warning Messages**

js

CopyEdit

console.warn("This is a warning!");

**Output:**

pgsql

CopyEdit

This is a warning!

*(Usually appears in yellow in the terminal.)*

**2. Advanced Console Methods**

**2.1 console.table() - Display Data in Table Format**

js

CopyEdit

const users = [

{ id: 1, name: "Alice", age: 25 },

{ id: 2, name: "Bob", age: 30 },

];

console.table(users);

**Output:**

pgsql

CopyEdit

┌─────────┬────┬────────┬─────┐

│ (index) │ id │ name │ age │

├─────────┼────┼────────┼─────┤

│ 0 │ 1 │ Alice │ 25 │

│ 1 │ 2 │ Bob │ 30 │

└─────────┴────┴────────┴─────┘

**2.2 console.time() and console.timeEnd() - Measure Execution Time**

js

CopyEdit

console.time("Loop Time");

for (let i = 0; i < 1000000; i++) {} // Some loop

console.timeEnd("Loop Time");

**Output (example):**

pgsql

CopyEdit

Loop Time: 5.432ms

**2.3 console.count() - Count the Number of Calls**

js

CopyEdit

console.count("Counter");

console.count("Counter");

console.count("Counter");

**Output:**

makefile

CopyEdit

Counter: 1

Counter: 2

Counter: 3

**2.4 console.group() and console.groupEnd() - Group Console Messages**

js

CopyEdit

console.group("User Info");

console.log("Name: Alice");

console.log("Age: 25");

console.groupEnd();

**Output:**

pgsql

CopyEdit

User Info

Name: Alice

Age: 25

**2.5 console.assert() - Log Only if Condition is False**

js

CopyEdit

const x = 5;

console.assert(x > 10, "x is not greater than 10!");

**Output:**

pgsql

CopyEdit

Assertion failed: x is not greater than 10!

*(Only logs if the condition fails.)*

**3. Using console in a Node.js File**

Create a file console-demo.js and add:

js

CopyEdit

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

console.warn("This is a warning message.");

console.error("This is an error message.");

console.table([{ name: "Alice", age: 25 }, { name: "Bob", age: 30 }]);

console.time("Execution Time");

for (let i = 0; i < 1000000; i++) {}

console.timeEnd("Execution Time");

Run it using:

sh

CopyEdit

node console-demo.js

**Conclusion**

The console module is a powerful debugging tool in Node.js. Use it effectively to log messages, measure performance, and debug efficiently.

**Node.js Command-Line Utilities**

Node.js provides built-in utilities to work with the command line. These are useful for building CLI tools, processing user input, and handling system operations.

**1. Running Node.js in the Command Line**

You can execute Node.js scripts using the command line.

**1.1 Run a Script File**

sh

CopyEdit

node script.js

Example script.js:

js

CopyEdit

console.log("Hello from Node.js!");

**1.2 Run Interactive Mode (REPL)**

Start the Node.js REPL (Read-Eval-Print Loop):

sh

CopyEdit

node

Then type:

js

CopyEdit

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

Exit REPL using:

sh

CopyEdit

.exit

or Ctrl + D

**2. Reading Command-Line Arguments (process.argv)**

**2.1 Basic Example**

Save the following in args.js:

js

CopyEdit

console.log(process.argv);

Run it with:

sh

CopyEdit

node args.js Hello World

**Output:**

bash

CopyEdit

[

'/usr/local/bin/node',

'/path/to/args.js',

'Hello',

'World'

]

Arguments start from index **2**.

**2.2 Extracting Arguments**

js

CopyEdit

const args = process.argv.slice(2);

console.log(`First argument: ${args[0]}`);

console.log(`Second argument: ${args[1]}`);

Run:

sh

CopyEdit

node args.js John Doe

**Output:**

sql

CopyEdit

First argument: John

Second argument: Doe

**3. Reading User Input (readline Module)**

js

CopyEdit

const readline = require("readline");

const rl = readline.createInterface({

input: process.stdin,

output: process.stdout,

});

rl.question("What is your name? ", (name) => {

console.log(`Hello, ${name}!`);

rl.close();

});

Run it:

sh

CopyEdit

node input.js

Example input/output:

pgsql

CopyEdit

What is your name? Alice

Hello, Alice!

**4. Using process for System Information**

**4.1 Get Environment Variables**

js

CopyEdit

console.log(process.env);

console.log(`User: ${process.env.USER}`);

**4.2 Get Current Working Directory**

js

CopyEdit

console.log(`Current Directory: ${process.cwd()}`);

**4.3 Get Node.js Version**

js

CopyEdit

console.log(`Node.js Version: ${process.version}`);

**5. File System Operations (fs Module)**

**5.1 Read a File**

js

CopyEdit

const fs = require("fs");

fs.readFile("test.txt", "utf8", (err, data) => {

if (err) throw err;

console.log(data);

});

**5.2 Write to a File**

js

CopyEdit

fs.writeFile("output.txt", "Hello, Node.js!", (err) => {

if (err) throw err;

console.log("File written!");

});

**6. Execute Shell Commands (child\_process Module)**

**6.1 Run a Shell Command**

js

CopyEdit

const { exec } = require("child\_process");

exec("ls", (err, stdout, stderr) => {

if (err) {

console.error(`Error: ${err}`);

return;

}

console.log(`Output: ${stdout}`);

});

*(Use dir instead of ls on Windows.)*

**6.2 Run a Script**

js

CopyEdit

exec("node script.js", (err, stdout) => {

console.log(stdout);

});

**7. Command-Line Arguments with yargs**

Install yargs:

sh

CopyEdit

npm install yargs

Example cli.js:

js

CopyEdit

const yargs = require("yargs");

const argv = yargs

.command("greet", "Prints a greeting", {

name: {

describe: "Your name",

demandOption: true,

type: "string",

},

})

.help()

.argv;

console.log(`Hello, ${argv.name}!`);

Run:

sh

CopyEdit

node cli.js greet --name=John

**Output:**

CopyEdit

Hello, John!

**8. Building a Simple CLI Tool**

Create calculator.js:

js

CopyEdit

const yargs = require("yargs");

const argv = yargs

.command("add", "Add two numbers", {

a: { describe: "First number", demandOption: true, type: "number" },

b: { describe: "Second number", demandOption: true, type: "number" },

})

.help()

.argv;

if (argv.\_[0] === "add") {

console.log(`Result: ${argv.a + argv.b}`);

}

Run:

sh

CopyEdit

node calculator.js add --a=5 --b=3

**Output:**

makefile

CopyEdit

Result: 8

**Conclusion**

Node.js provides powerful command-line utilities to:

* Read arguments (process.argv)
* Handle user input (readline)
* Execute shell commands (child\_process)
* Read/write files (fs)
* Build CLI tools (yargs)

**Node.js Modules**

Modules in Node.js are **reusable blocks of code** that help keep applications modular and organized. They can be **built-in (core modules), third-party (npm modules), or custom modules**.

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

1️⃣ **Core Modules** – Pre-built in Node.js (e.g., fs, http, path).  
2️⃣ **Local (Custom) Modules** – User-defined JavaScript files.  
3️⃣ **Third-Party Modules** – Installed via npm (e.g., express, lodash).

**2. Core (Built-in) Modules**

Node.js comes with several built-in modules. To use them, require the module:

**2.1 fs (File System) – Read and Write Files**

js

CopyEdit

const fs = require("fs");

// Write to a file

fs.writeFileSync("test.txt", "Hello, Node.js!");

// Read from a file

const data = fs.readFileSync("test.txt", "utf8");

console.log(data);

**2.2 http – Create a Simple Web Server**

js

CopyEdit

const http = require("http");

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

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

res.end("Hello, World!");

});

server.listen(3000, () => console.log("Server running on http://localhost:3000"));

**2.3 path – Work with File Paths**

js

CopyEdit

const path = require("path");

console.log(path.basename("/users/admin/file.txt")); // Output: file.txt

console.log(path.dirname("/users/admin/file.txt")); // Output: /users/admin

console.log(path.extname("/users/admin/file.txt")); // Output: .txt

**2.4 os – Get System Information**

js

CopyEdit

const os = require("os");

console.log(os.platform()); // Output: win32, linux, darwin (Mac)

console.log(os.arch()); // Output: x64, arm, etc.

console.log(os.freemem()); // Output: Free memory in bytes

**3. Custom (Local) Modules**

**3.1 Creating a Module**

Create math.js:

js

CopyEdit

exports.add = (a, b) => a + b;

exports.sub = (a, b) => a - b;

**3.2 Import and Use in Another File**

Create app.js:

js

CopyEdit

const math = require("./math");

console.log(math.add(5, 3)); // Output: 8

console.log(math.sub(5, 3)); // Output: 2

**4. Third-Party Modules (npm)**

**4.1 Installing a Module**

Example: Install lodash (a utility library)

sh

CopyEdit

npm install lodash

**4.2 Using lodash in a Script**

js

CopyEdit

const \_ = require("lodash");

const arr = [1, 2, 3, 4, 5];

console.log(\_.shuffle(arr)); // Randomly shuffles array elements

**5. Exporting Multiple Functions (Custom Module)**

**5.1 Using module.exports**

Create calculator.js:

js

CopyEdit

const add = (a, b) => a + b;

const subtract = (a, b) => a - b;

module.exports = { add, subtract };

**5.2 Importing in app.js**

js

CopyEdit

const calculator = require("./calculator");

console.log(calculator.add(10, 5)); // Output: 15

console.log(calculator.subtract(10, 5)); // Output: 5

**6. Importing ES6 Modules (import/export)**

**6.1 Enabling ES6 Modules**

* Add "type": "module" in package.json
* Use import and export

**6.2 Create math.mjs**

js

CopyEdit

export function add(a, b) {

return a + b;

}

export function sub(a, b) {

return a - b;

}

**6.3 Importing in Another File**

js

CopyEdit

import { add, sub } from "./math.mjs";

console.log(add(5, 3)); // Output: 8

console.log(sub(5, 3)); // Output: 2

**Conclusion**

✔ **Core Modules** – Pre-built in Node.js (fs, http, path, etc.).  
✔ **Custom Modules** – Your own functions exported/imported using require or import.  
✔ **Third-Party Modules** – Installed via npm (express, lodash, etc.).

**Node JS Concepts**

Node.js is a runtime environment for executing JavaScript code outside a web browser. It is asynchronous, event-driven, and non-blocking, making it great for building scalable network applications.

Node.js Architecture

* Single-Threaded – Uses a single main thread for execution.
* Event-Driven – Uses events and callbacks instead of waiting for processes to complete.
* Non-Blocking I/O – Handles multiple requests without blocking execution.
* V8 Engine – Uses Google's V8 JavaScript engine to execute code efficiently.

Example of Non-Blocking I/O:

js

CopyEdit

const fs = require("fs");

console.log("Start");

fs.readFile("test.txt", "utf8", (err, data) => {

if (err) throw err;

console.log(data);

});

console.log("End");

Output:

sql

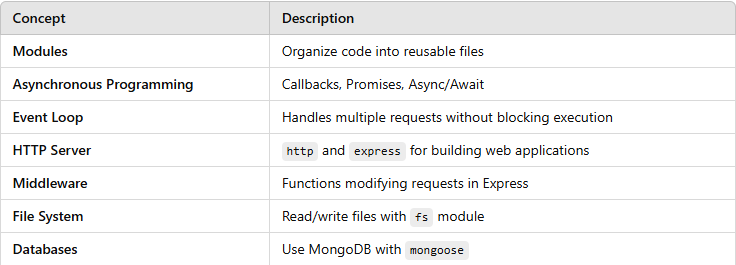
CopyEdit

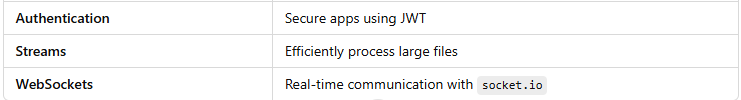
Start

End

(File contents printed later)

Even though fs.readFile() takes time, Node.js doesn’t wait for it—it moves on to the next operation.





**Node.js Events**

Node.js uses an event-driven architecture, making it efficient and scalable for handling asynchronous operations.

1. What are Events in Node.js?

Events in Node.js are similar to JavaScript events (like click or keydown) but work on the server-side.

* The EventEmitter class in Node.js allows you to create, listen to, and emit events.

How Events Work in Node.js

1️⃣ Create an event emitter.  
2️⃣ Listen for an event.  
3️⃣ Emit the event when needed.

2. Importing events Module

Node.js provides the events module to handle event-driven operations.

js

CopyEdit

const EventEmitter = require("events"); // Import the module

const eventEmitter = new EventEmitter(); // Create an event emitter

3. Basic Event Handling

3.1 Creating and Emitting Events

js

CopyEdit

const EventEmitter = require("events");

const event = new EventEmitter();

// Register an event listener

event.on("greet", (name) => {

console.log(`Hello, ${name}!`);

});

// Emit the event

event.emit("greet", "Alice");

Output:

CopyEdit

Hello, Alice!

4. Handling Multiple Listeners

You can register multiple listeners for the same event.

js

CopyEdit

event.on("greet", () => console.log("How are you?"));

event.emit("greet", "Bob");

Output:

sql

CopyEdit

Hello, Bob!

How are you?

5. Passing Multiple Arguments

js

CopyEdit

event.on("user", (name, age) => {

console.log(`User: ${name}, Age: ${age}`);

});

event.emit("user", "John", 30);

Output:

yaml

CopyEdit

User: John, Age: 30

6. once() – Execute Event Only Once

The once() method ensures the listener executes only once.

js

CopyEdit

event.once("message", () => {

console.log("This will run only once.");

});

event.emit("message");

event.emit("message"); // Won't execute again

Output:

arduino

CopyEdit

This will run only once.

7. Removing Event Listeners

7.1 Remove a Specific Listener

js

CopyEdit

const greet = (name) => console.log(`Hello, ${name}!`);

event.on("greet", greet);

event.off("greet", greet); // Remove listener

event.emit("greet", "Alice"); // No output

7.2 Remove All Listeners

js

CopyEdit

event.removeAllListeners("greet");

8. Checking the Number of Listeners

js

CopyEdit

console.log(event.listenerCount("greet"));

9. Practical Use Case – Event-Driven File System

js

CopyEdit

const fs = require("fs");

const event = new EventEmitter();

// Event to read a file

event.on("readFile", (file) => {

fs.readFile(file, "utf8", (err, data) => {

if (err) return console.error(err);

console.log(`File Content: ${data}`);

});

});

// Emit the event

event.emit("readFile", "test.txt");

10. Using Events with HTTP Server

js

CopyEdit

const http = require("http");

const EventEmitter = require("events");

const event = new EventEmitter();

event.on("request", (req, res) => {

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

res.end("Hello from the event-driven server!");

});

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

event.emit("request", req, res);

});

server.listen(3000, () => console.log("Server running on http://localhost:3000"));

**Summary**

| **Method** | **Description** |
| --- | --- |
| on(event, listener) | Register an event listener |
| emit(event, ...args) | Trigger an event |
| once(event, listener) | Listen for an event only once |
| off(event, listener) | Remove a specific listener |
| removeAllListeners(event) | Remove all listeners for an event |
| listenerCount(event) | Get the number of listeners |

Node.js events power real-time applications like chat apps, notifications, and file system monitoring.

**Node.js Database Access**

Node.js supports multiple databases, including **SQL (MySQL, PostgreSQL)** and **NoSQL (MongoDB, Firebase, Redis)**. Below are steps to connect and interact with different databases.

**1. MySQL (Relational Database)**

**1.1 Install MySQL Module**

sh

CopyEdit

npm install mysql2

**1.2 Connect to MySQL**

js

CopyEdit

const mysql = require("mysql2");

const connection = mysql.createConnection({

host: "localhost",

user: "root",

password: "password",

database: "testdb",

});

connection.connect((err) => {

if (err) throw err;

console.log("Connected to MySQL!");

});

**1.3 Create a Table**

js

CopyEdit

const sql = `CREATE TABLE users (id INT AUTO\_INCREMENT PRIMARY KEY, name VARCHAR(255), age INT)`;

connection.query(sql, (err, result) => {

if (err) throw err;

console.log("Table created");

});

**1.4 Insert Data**

js

CopyEdit

const sql = "INSERT INTO users (name, age) VALUES (?, ?)";

connection.query(sql, ["Alice", 25], (err, result) => {

if (err) throw err;

console.log("User inserted!");

});

**1.5 Fetch Data**

js

CopyEdit

connection.query("SELECT \* FROM users", (err, results) => {

if (err) throw err;

console.log(results);

});

**2. PostgreSQL (Relational Database)**

**2.1 Install PostgreSQL Module**

sh

CopyEdit

npm install pg

**2.2 Connect to PostgreSQL**

js

CopyEdit

const { Client } = require("pg");

const client = new Client({

user: "postgres",

host: "localhost",

database: "testdb",

password: "password",

port: 5432,

});

client.connect()

.then(() => console.log("Connected to PostgreSQL"))

.catch(err => console.error(err));

**2.3 Create a Table**

js

CopyEdit

client.query(`CREATE TABLE users (id SERIAL PRIMARY KEY, name VARCHAR(100), age INT)`)

.then(() => console.log("Table created"))

.catch(err => console.error(err));

**2.4 Insert Data**

js

CopyEdit

client.query("INSERT INTO users (name, age) VALUES ($1, $2)", ["Bob", 30])

.then(() => console.log("User added"))

.catch(err => console.error(err));

**2.5 Fetch Data**

js

CopyEdit

client.query("SELECT \* FROM users")

.then(res => console.log(res.rows))

.catch(err => console.error(err));

**3. MongoDB (NoSQL Database)**

**3.1 Install MongoDB and Mongoose**

sh

CopyEdit

npm install mongoose

**3.2 Connect to MongoDB**

js

CopyEdit

const mongoose = require("mongoose");

mongoose.connect("mongodb://localhost:27017/testdb", { useNewUrlParser: true, useUnifiedTopology: true })

.then(() => console.log("Connected to MongoDB"))

.catch(err => console.error(err));

**3.3 Define Schema & Model**

js

CopyEdit

const userSchema = new mongoose.Schema({

name: String,

age: Number

});

const User = mongoose.model("User", userSchema);

**3.4 Insert Data**

js

CopyEdit

const user = new User({ name: "Charlie", age: 28 });

user.save()

.then(() => console.log("User saved!"))

.catch(err => console.error(err));

**3.5 Fetch Data**

js

CopyEdit

User.find()

.then(users => console.log(users))

.catch(err => console.error(err));

**4. Firebase (NoSQL Cloud Database)**

**4.1 Install Firebase SDK**

sh

CopyEdit

npm install firebase-admin

**4.2 Setup Firebase**

js

CopyEdit

const admin = require("firebase-admin");

const serviceAccount = require("./firebase-key.json");

admin.initializeApp({

credential: admin.credential.cert(serviceAccount),

databaseURL: "https://your-database.firebaseio.com"

});

const db = admin.firestore();

**4.3 Insert Data**

js

CopyEdit

db.collection("users").add({ name: "Eve", age: 27 })

.then(() => console.log("User added"))

.catch(err => console.error(err));

**4.4 Fetch Data**

js

CopyEdit

db.collection("users").get()

.then(snapshot => {

snapshot.forEach(doc => console.log(doc.data()));

})

.catch(err => console.error(err));

**5. Redis (In-Memory Database)**

**5.1 Install Redis**

sh

CopyEdit

npm install redis

**5.2 Connect to Redis**

js

CopyEdit

const redis = require("redis");

const client = redis.createClient();

client.on("connect", () => {

console.log("Connected to Redis");

});

**5.3 Store Data**

js

CopyEdit

client.set("username", "Alice", redis.print);

**5.4 Retrieve Data**

js

CopyEdit

client.get("username", (err, reply) => {

console.log(reply);

});

**6. Database Comparison**

| **Feature** | **MySQL/PostgreSQL (SQL)** | **MongoDB/Firebase (NoSQL)** | **Redis (In-Memory)** |
| --- | --- | --- | --- |
| Data Structure | Tables & Rows | JSON Documents | Key-Value Pairs |
| Query Language | SQL | Mongo Query / NoSQL | Commands |
| Scalability | Vertical | Horizontal | High-Speed |
| Transactions | Strong | Limited | Limited |

**Conclusion**

✔ **MySQL/PostgreSQL** – Best for structured data & transactions.  
✔ **MongoDB/Firebase** – Best for unstructured, scalable NoSQL data.  
✔ **Redis** – Best for caching and fast access.

**Node.js with Express.js**

Express.js is a **fast, minimalist framework** for building web applications in Node.js. It simplifies routing, middleware, and handling requests & responses.

**1. Install Express.js**

sh

CopyEdit

npm install express

**2. Create a Basic Express Server**

js

CopyEdit

const express = require("express");

const app = express();

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

res.send("Hello, Express!");

});

app.listen(3000, () => console.log("Server running on http://localhost:3000"));

🔹 **Run the server:** node app.js  
🔹 **Open in browser:** http://localhost:3000

**3. Express Routing**

**3.1 Handling Different HTTP Methods**

js

CopyEdit

app.get("/users", (req, res) => res.send("GET request"));

app.post("/users", (req, res) => res.send("POST request"));

app.put("/users/:id", (req, res) => res.send(`PUT request for ID: ${req.params.id}`));

app.delete("/users/:id", (req, res) => res.send(`DELETE request for ID: ${req.params.id}`));

**4. Middleware in Express**

Middleware functions process requests **before they reach the response**.

**4.1 Example of Middleware**

js

CopyEdit

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

console.log(`Request Method: ${req.method}, URL: ${req.url}`);

next();

});

**4.2 Using express.json() for Parsing JSON Requests**

js

CopyEdit

app.use(express.json());

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

res.send(`Received: ${JSON.stringify(req.body)}`);

});

**5. Handling Static Files**

js

CopyEdit

app.use(express.static("public")); // Serves files from the "public" folder

**6. Express with Template Engines (EJS)**

**6.1 Install EJS**

sh

CopyEdit

npm install ejs

**6.2 Set View Engine**

js

CopyEdit

app.set("view engine", "ejs");

**6.3 Render an EJS File**

js

CopyEdit

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

res.render("profile", { name: "Alice", age: 25 });

});

🔹 Create views/profile.ejs:

html

CopyEdit

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

<p>Age: <%= age %></p>

**7. Express with MongoDB**

**7.1 Install Mongoose**

sh

CopyEdit

npm install mongoose

**7.2 Connect to MongoDB**

js

CopyEdit

const mongoose = require("mongoose");

mongoose.connect("mongodb://localhost:27017/testdb", { useNewUrlParser: true, useUnifiedTopology: true });

const UserSchema = new mongoose.Schema({ name: String, age: Number });

const User = mongoose.model("User", UserSchema);

**7.3 API with MongoDB**

js

CopyEdit

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

const user = new User(req.body);

await user.save();

res.send("User saved!");

});

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

const users = await User.find();

res.json(users);

});

**8. Express with JWT Authentication**

**8.1 Install JWT**

sh

CopyEdit

npm install jsonwebtoken

**8.2 Generate Token**

js

CopyEdit

const jwt = require("jsonwebtoken");

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

const user = { id: 1, name: "Alice" };

const token = jwt.sign(user, "secretkey", { expiresIn: "1h" });

res.json({ token });

});

**8.3 Protect Routes**

js

CopyEdit

const verifyToken = (req, res, next) => {

const token = req.headers["authorization"];

if (!token) return res.status(403).send("Token required");

jwt.verify(token, "secretkey", (err, decoded) => {

if (err) return res.status(401).send("Invalid token");

req.user = decoded;

next();

});

};

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

res.send("Protected route accessed");

});

**9. Express with WebSockets (Real-Time Chat)**

**9.1 Install Socket.io**

sh

CopyEdit

npm install socket.io

**9.2 Setup WebSocket Server**

js

CopyEdit

const http = require("http");

const { Server } = require("socket.io");

const server = http.createServer(app);

const io = new Server(server);

io.on("connection", (socket) => {

console.log("User connected");

socket.on("message", (msg) => io.emit("message", msg));

});

server.listen(3000, () => console.log("Server running on http://localhost:3000"));

**10. Express Project Structure (MVC)**

bash

CopyEdit

/myapp

│── app.js # Main server file

│── /routes

│ ├── users.js # User routes

│── /models

│ ├── User.js # User schema

│── /controllers

│ ├── userController.js # User logic

│── /views

│ ├── index.ejs # Template file

**Example Route (routes/users.js)**

js

CopyEdit

const express = require("express");

const router = express.Router();

const User = require("../models/User");

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

const users = await User.find();

res.json(users);

});

module.exports = router;

**Use in app.js**

js

CopyEdit

const userRoutes = require("./routes/users");

app.use("/users", userRoutes);

**Summary**

| **Feature** | **Code Example** |
| --- | --- |
| **Basic Server** | app.listen(3000, () => console.log("Server running")) |
| **Routes** | app.get("/", (req, res) => res.send("Hello")) |
| **Middleware** | app.use(express.json()) |
| **Static Files** | app.use(express.static("public")) |
| **MongoDB Integration** | mongoose.connect("mongodb://localhost:27017/db") |
| **JWT Authentication** | jwt.sign(payload, secret, {expiresIn: "1h"}) |
| **WebSockets** | const io = new Server(server) |

**Express.js Requests**

In Express.js, handling **HTTP requests** (GET, POST, PUT, DELETE) is straightforward. Requests contain **headers, body, params, and query strings**.

**1. Handling GET Requests**

🔹 GET requests **fetch data** and don’t send a request body.

**Example: Basic GET Route**

js

CopyEdit

const express = require("express");

const app = express();

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

res.send("Welcome to Express!");

});

app.listen(3000, () => console.log("Server running on http://localhost:3000"));

✅ **Access:** http://localhost:3000/  
✅ **Response:** "Welcome to Express!"

**2. Handling Query Parameters (req.query)**

🔹 Used to pass **optional data** in URLs (?key=value).

**Example: Using Query Parameters**

js

CopyEdit

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

const { name, age } = req.query;

res.send(`Search for Name: ${name}, Age: ${age}`);

});

✅**RequestURL:**http://localhost:3000/search?name=Alice&age=25  
✅ **Response:** "Search for Name: Alice, Age: 25"

**3. Handling URL Parameters (req.params)**

🔹 Used for **dynamic routing** (/users/:id).

**Example: Using Route Parameters**

js

CopyEdit

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

res.send(`User ID: ${req.params.id}`);

});

✅**RequestURL:**http://localhost:3000/users/101  
✅ **Response:** "User ID: 101"

**4. Handling POST Requests (req.body)**

🔹 POST requests send **data in the request body**.  
🔹 **Enable JSON parsing** with express.json().

**Example: Handling JSON Data in POST Request**

js

CopyEdit

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

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

const { name, age } = req.body;

res.send(`User created: ${name}, Age: ${age}`);

});

✅ **POST Request Body (JSON):**

json

CopyEdit

{

"name": "Bob",

"age": 30

}

✅ **Response:** "User created: Bob, Age: 30"

**5. Handling PUT Requests (Update Data)**

🔹 Used to **update existing resources**.

**Example: Handling PUT Request**

js

CopyEdit

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

res.send(`User ${req.params.id} updated`);

});

✅**Request:**PUThttp://localhost:3000/users/101  
✅ **Response:** "User 101 updated"

**6. Handling DELETE Requests (Delete Data)**

🔹 Used to **delete resources**.

**Example: Handling DELETE Request**

js

CopyEdit

app.delete("/users/:id", (req, res) => {

res.send(`User ${req.params.id} deleted`);

});

✅**Request:**DELETE http://localhost:3000/users/101  
✅ **Response:** "User 101 deleted"

**7. Handling Request Headers (req.headers)**

🔹 Access request headers using req.headers.

**Example: Get Headers**

js

CopyEdit

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

console.log(req.headers);

res.send("Check server console for headers");

});

✅ **Client Request (cURL Example):**

sh

CopyEdit

curl -H "Authorization: Bearer abc123" http://localhost:3000/headers

✅ **Access Header in Code:**

js

CopyEdit

const authHeader = req.headers["authorization"];

console.log(`Authorization: ${authHeader}`);

**8. Uploading Files in Express (multer)**

🔹 **Install multer:** npm install multer

**Example: File Upload API**

js

CopyEdit

const multer = require("multer");

const upload = multer({ dest: "uploads/" });

app.post("/upload", upload.single("file"), (req, res) => {

res.send(`File uploaded: ${req.file.originalname}`);

});

✅**Request:**Send a file as file in a POST request  
✅ **Response:** "File uploaded: filename.jpg"

**9. Full CRUD API Example**

js

CopyEdit

const express = require("express");

const app = express();

app.use(express.json());

let users = [{ id: 1, name: "Alice" }];

app.get("/users", (req, res) => res.json(users));

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

const user = { id: users.length + 1, ...req.body };

users.push(user);

res.json(user);

});

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

const user = users.find(u => u.id == req.params.id);

if (!user) return res.status(404).send("User not found");

user.name = req.body.name;

res.json(user);

});

app.delete("/users/:id", (req, res) => {

users = users.filter(u => u.id != req.params.id);

res.send("User deleted");

});

app.listen(3000, () => console.log("Server running on port 3000"));

**Summary Table**

| **HTTP Method** | **Description** | **Access Data** |
| --- | --- | --- |
| **GET** | Fetch data | req.query, req.params |
| **POST** | Send data | req.body |
| **PUT** | Update data | req.params, req.body |
| **DELETE** | Remove data | req.params |
| **HEADERS** | Get metadata | req.headers |

**Key Takeaways**

✅Use req.query for optional query parameters.  
✅Use req.params for required route parameters.  
✅Use req.body for sending JSON data (Enable express.json()).  
✅ Use req.headers to access request headers.

**Express.js Response Handling**

In Express.js, the res object is used to **send responses** to clients. It includes methods like res.send(), res.json(), res.status(), res.redirect(), and more.

**1. Sending a Basic Response (res.send())**

🔹 Used for sending **plain text, HTML, or other data**.

**Example: Sending a String Response**

js

CopyEdit

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

res.send("Hello, Express!");

});

✅ **Response:** "Hello, Express!"

**Example: Sending an HTML Response**

js

CopyEdit

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

res.send("<h1>Welcome to Express</h1>");

});

✅ **Response:** <h1>Welcome to Express</h1>

**2. Sending JSON Response (res.json())**

🔹 Sends a **JSON response** (useful for APIs).

**Example: Sending JSON Data**

js

CopyEdit

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

res.json({ name: "Alice", age: 25 });

});

✅ **Response (JSON):**

json

CopyEdit

{

"name": "Alice",

"age": 25

}

✅ **Difference between res.send() and res.json()**

* res.json() automatically sets Content-Type: application/json.
* res.send() can send JSON but doesn’t enforce the content type.

**3. Setting HTTP Status Codes (res.status())**

🔹 Used to set the HTTP status code before sending a response.

**Example: Sending a 404 Not Found Response**

js

CopyEdit

app.get("/not-found", (req, res) => {

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

});

✅ **Response:** "Page not found" (HTTP 404)

**Example: Sending JSON with a Status Code**

js

CopyEdit

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

res.status(500).json({ error: "Internal Server Error" });

});

✅ **Response (HTTP 500):**

json

CopyEdit

{

"error": "Internal Server Error"

}

**4. Redirecting Requests (res.redirect())**

🔹 Redirects the client to another URL.

**Example: Redirect to Another Page**

js

CopyEdit

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

res.redirect("/new-route");

});

✅**Access:**http://localhost:3000/old-route  
✅ **Redirects to:** http://localhost:3000/new-route

**Example: Redirect to an External URL**

js

CopyEdit

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

res.redirect("https://www.google.com");

});

✅**Access:**http://localhost:3000/google  
✅ **Redirects to:** Google

**5. Sending Files (res.sendFile())**

🔹 Used to send files to clients.

**Example: Sending an HTML File**

js

CopyEdit

const path = require("path");

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

res.sendFile(path.join(\_\_dirname, "index.html"));

});

✅ **Response:** Serves index.html from the server.

**6. Setting Response Headers (res.set())**

🔹 Used to modify response headers.

**Example: Custom Headers**

js

CopyEdit

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

res.set("X-Custom-Header", "Hello");

res.send("Header set!");

});

✅ **Response Header:**

css

CopyEdit

X-Custom-Header: Hello

**7. Sending Cookies (res.cookie())**

🔹 Used to set cookies in the client’s browser.

**Example: Setting a Cookie**

js

CopyEdit

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

res.cookie("username", "Alice");

res.send("Cookie set!");

});

✅ **Response Header:**

javascript

CopyEdit

Set-Cookie: username=Alice

**Example: Clearing a Cookie**

js

CopyEdit

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

res.clearCookie("username");

res.send("Cookie cleared!");

});

✅ **Clears the username cookie.**

**8. Streaming Response (res.write())**

🔹 Used for sending **chunked responses**.

**Example: Sending a Streamed Response**

js

CopyEdit

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

res.write("Hello ");

setTimeout(() => res.write("World!"), 1000);

setTimeout(() => res.end(), 2000);

});

✅ **Response (streamed in chunks):** "Hello " → "World!"

**9. Ending a Response (res.end())**

🔹 Explicitly **closes the response**.

**Example: Manually Ending a Response**

js

CopyEdit

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

res.write("Processing...");

res.end();

});

✅ **Response:** "Processing..."

**10. Full API Example**

js

CopyEdit

const express = require("express");

const app = express();

app.use(express.json());

app.get("/", (req, res) => res.send("Welcome to Express!"));

app.get("/json", (req, res) => res.json({ message: "Hello JSON" }));

app.get("/status", (req, res) => res.status(403).send("Forbidden"));

app.get("/redirect", (req, res) => res.redirect("https://example.com"));

app.get("/file", (req, res) => res.sendFile(\_\_dirname + "/index.html"));

app.get("/cookie", (req, res) => res.cookie("user", "Alice").send("Cookie set"));

app.post("/data", (req, res) => res.json({ received: req.body }));

app.use((req, res) => res.status(404).send("Page not found"));

app.listen(3000, () => console.log("Server running on port 3000"));

**Summary Table**

| **Method** | **Description** | **Example** |
| --- | --- | --- |
| res.send() | Send text/HTML | res.send("Hello") |
| res.json() | Send JSON response | res.json({name: "Alice"}) |
| res.status() | Set HTTP status | res.status(404).send("Not Found") |
| res.redirect() | Redirect to URL | res.redirect("/home") |
| res.sendFile() | Send a file | res.sendFile("/path/to/file") |
| res.set() | Set custom headers | res.set("X-Key", "Value") |
| res.cookie() | Set a cookie | res.cookie("user", "Alice") |
| res.clearCookie() | Clear a cookie | res.clearCookie("user") |
| res.write() | Send streaming data | res.write("Part 1") |
| res.end() | End response manually | res.end() |

**Key Takeaways**

✅res.send() for text, res.json() for API responses.  
✅Use res.status() to control HTTP response codes.  
✅Use res.redirect() for URL redirections.  
✅res.cookie() and res.clearCookie() manage cookies.  
✅ res.sendFile() serves static files.

**Express.js GET Requests**

In **Express.js**, a GET request is used to **fetch data** from the server. It is commonly used for retrieving information from a database, API, or file system.

**1. Basic GET Route**

🔹 app.get(path, callback) defines a route that listens for GET requests.  
🔹 req (request) contains **incoming data** (query params, headers, etc.).  
🔹 res (response) sends data **back to the client**.

**Example: Basic GET Request**

js

CopyEdit

const express = require("express");

const app = express();

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

res.send("Welcome to Express!");

});

app.listen(3000, () => console.log("Server running on http://localhost:3000"));

✅**AccessURL:**http://localhost:3000/  
✅ **Response:** "Welcome to Express!"

**2. Handling Query Parameters (req.query)**

🔹Query parameters are **optional values** passed in the URL after ?.  
🔹Multiple parameters are separated using &.  
🔹 Access them using req.query.

**Example: Query Parameters**

js

CopyEdit

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

const { keyword, page } = req.query;

res.send(`Search results for "${keyword}" on page ${page}`);

});

✅ **Request URL:**

bash

CopyEdit

http://localhost:3000/search?keyword=NodeJS&page=2

✅ **Response:**

nginx

CopyEdit

Search results for "NodeJS" on page 2

**3. Handling URL Parameters (req.params)**

🔹Used to **capture dynamic values** from the URL.  
🔹 Defined using : before the parameter name.

**Example: Route Parameters**

js

CopyEdit

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

res.send(`User ID: ${req.params.id}`);

});

✅**RequestURL:**http://localhost:3000/users/101  
✅ **Response:** "User ID: 101"

**4. Handling Headers (req.headers)**

🔹 Access request headers using req.headers.

**Example: Getting Headers**

js

CopyEdit

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

console.log(req.headers);

res.send("Check server console for headers");

});

✅ **Client Request (cURL Example):**

sh

CopyEdit

curl -H "Authorization: Bearer abc123" http://localhost:3000/headers

✅ **Access Header in Code:**

js

CopyEdit

const authHeader = req.headers["authorization"];

console.log(`Authorization: ${authHeader}`);

**5. Sending JSON Response (res.json())**

🔹 JSON is commonly used in APIs to send structured data.

**Example: Sending JSON Data**

js

CopyEdit

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

res.json({ name: "Alice", age: 25 });

});

✅ **Response (JSON):**

json

CopyEdit

{

"name": "Alice",

"age": 25

}

**6. Handling Multiple URL Parameters**

🔹 You can define multiple parameters in a single route.

**Example: Multiple URL Parameters**

js

CopyEdit

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

res.send(`Category: ${req.params.category}, Post ID: ${req.params.id}`);

});

✅**RequestURL:**http://localhost:3000/posts/tech/500  
✅ **Response:** "Category: tech, Post ID: 500"

**7. Using Middleware in GET Requests**

🔹 Middleware functions process requests **before reaching the route handler**.

**Example: Middleware in a GET Request**

js

CopyEdit

const logger = (req, res, next) => {

console.log(`Request received at: ${new Date().toISOString()}`);

next(); // Move to the next middleware/route

};

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

res.send("Information page");

});

✅ **Logs timestamp in the console when /info is accessed.**

**8. Using Async/Await in GET Requests**

🔹 Useful when fetching data from databases or external APIs.

**Example: GET Request with Async Function**

js

CopyEdit

const fetchData = async () => {

return { message: "Data retrieved successfully" };

};

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

const data = await fetchData();

res.json(data);

});

✅ **Response (JSON):**

json

CopyEdit

{

"message": "Data retrieved successfully"

}

**9. Handling 404 for Unknown Routes**

🔹 Use app.use() to handle undefined routes.

**Example: Handling Unknown Routes**

js

CopyEdit

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

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

});

✅ **Accessing a non-existent route returns:** "Page not found" (HTTP 404).

**10. Full Example (GET Routes with Different Methods)**

js

CopyEdit

const express = require("express");

const app = express();

// Basic GET request

app.get("/", (req, res) => res.send("Welcome to Express!"));

// Query parameters

app.get("/search", (req, res) => res.send(`Search keyword: ${req.query.keyword}`));

// URL parameters

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

// JSON response

app.get("/json", (req, res) => res.json({ message: "Hello, JSON!" }));

// Middleware example

const logRequest = (req, res, next) => {

console.log(`Request made to: ${req.path}`);

next();

};

app.get("/middleware", logRequest, (req, res) => res.send("Middleware applied"));

// Handle unknown routes

app.use((req, res) => res.status(404).send("Route not found"));

app.listen(3000, () => console.log("Server running on http://localhost:3000"));

**Summary Table**

| **Feature** | **Method** | **Example** |
| --- | --- | --- |
| **Basic GET** | app.get() | res.send("Hello") |
| **Query Params** | req.query | req.query.name |
| **URL Params** | req.params | req.params.id |
| **Headers** | req.headers | req.headers["key"] |
| **JSON Response** | res.json() | res.json({ name: "Alice" }) |
| **Redirect** | res.redirect() | res.redirect("/home") |
| **Middleware** | next() | app.use(logger) |

**📌 Key Takeaways**

✅Use req.query for optional query parameters.  
✅Use req.params for required route parameters.  
✅Use res.json() for structured API responses.  
✅Use middleware (next()) for logging or authentication.  
✅ Handle unknown routes with a 404 response.

**Express.js POST Requests**

In **Express.js**, a POST request is used to **send data to the server**. It is commonly used for creating or updating resources in a database.

**1. Setting Up a Basic POST Route**

🔹app.post(path, callback) defines a route that listens for POST requests.  
🔹req.body contains the incoming **data from the client**.  
🔹 Requires **middleware (express.json())** to parse JSON data.

**Example: Basic POST Request**

js

CopyEdit

const express = require("express");

const app = express();

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

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

res.send("POST request received!");

});

app.listen(3000, () => console.log("Server running on http://localhost:3000"));

✅ **Test with cURL:**

sh

CopyEdit

curl -X POST http://localhost:3000/submit

✅ **Response:** "POST request received!"

**2. Handling JSON Data in POST Requests**

🔹 req.body contains the **sent JSON data**.  
🔹 express.json() must be enabled to **parse JSON payloads**.

**Example: Receiving JSON Data**

js

CopyEdit

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

console.log(req.body); // Logs incoming data

res.json({ message: "Data received!", data: req.body });

});

✅ **Test with cURL:**

sh

CopyEdit

curl -X POST http://localhost:3000/data -H "Content-Type: application/json" -d '{"name":"Alice","age":25}'

✅ **Response:**

json

CopyEdit

{

"message": "Data received!",

"data": {

"name": "Alice",

"age": 25

}

}

**3. Handling Form Data (express.urlencoded())**

🔹 When sending **form data** (application/x-www-form-urlencoded), use express.urlencoded().

**Example: Handling Form Data**

js

CopyEdit

app.use(express.urlencoded({ extended: true })); // Middleware for form data

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

res.send(`Received form data: Name - ${req.body.name}, Age - ${req.body.age}`);

});

✅ **Test with cURL:**

sh

CopyEdit

curl -X POST http://localhost:3000/form -d "name=Bob&age=30"

✅ **Response:**

css

CopyEdit

Received form data: Name - Bob, Age - 30

**4. Handling URL Parameters in POST Requests**

🔹 You can capture dynamic values from the URL using req.params.

**Example: Handling URL Parameters**

js

CopyEdit

app.post("/users/:id", (req, res) => {

res.send(`User ID: ${req.params.id}`);

});

✅ **Test with cURL:**

sh

CopyEdit

curl -X POST http://localhost:3000/users/101

✅ **Response:** "User ID: 101"

**5. Handling Headers in POST Requests**

🔹 Access request headers using req.headers.

**Example: Reading Headers**

js

CopyEdit

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

const token = req.headers["authorization"];

res.send(`Received Token: ${token}`);

});

✅ **Test with cURL:**

sh

CopyEdit

curl -X POST http://localhost:3000/headers -H "Authorization: Bearer abc123"

✅ **Response:** "Received Token: Bearer abc123"

**6. Sending a Custom Status Code**

🔹 Use res.status(code) to return custom HTTP status codes.

**Example: Custom Status Code**

js

CopyEdit

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

res.status(201).json({ message: "Resource created!" });

});

✅ **Response (201 Created):**

json

CopyEdit

{

"message": "Resource created!"

}

**7. Redirecting in POST Requests**

🔹 Use res.redirect(url) to redirect after a successful POST request.

**Example: Redirect After POST**

js

CopyEdit

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

res.redirect("/success");

});

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

res.send("Successfully redirected!");

});

✅ **Response:** "Successfully redirected!"

**8. Using Middleware for POST Requests**

🔹 Middleware can validate, log, or process request data **before** it reaches the handler.

**Example: Logging Middleware**

js

CopyEdit

const logRequest = (req, res, next) => {

console.log(`POST Request to: ${req.path}`);

next();

};

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

res.send("Middleware executed!");

});

✅ **Console Output:**

bash

CopyEdit

POST Request to: /log

✅ **Response:** "Middleware executed!"

**9. Handling File Uploads in POST Requests**

🔹 Use multer for handling file uploads.

**Install Multer:**

sh

CopyEdit

npm install multer

**Example: File Upload Handling**

js

CopyEdit

const multer = require("multer");

const upload = multer({ dest: "uploads/" }); // Store files in 'uploads/' directory

app.post("/upload", upload.single("file"), (req, res) => {

res.send(`File uploaded: ${req.file.originalname}`);

});

✅ **Test File Upload (cURL):**

sh

CopyEdit

curl -X POST http://localhost:3000/upload -F "file=@/path/to/file.jpg"

✅ **Response:** "File uploaded: file.jpg"

**10. Full Express.js POST API Example**

js

CopyEdit

const express = require("express");

const multer = require("multer");

const app = express();

const upload = multer({ dest: "uploads/" });

app.use(express.json());

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

// Basic POST request

app.post("/", (req, res) => res.send("Welcome to Express POST!"));

// JSON request handling

app.post("/json", (req, res) => res.json({ message: "Received JSON", data: req.body }));

// Form data handling

app.post("/form", (req, res) => res.send(`Received: ${req.body.name}, ${req.body.age}`));

// Handling URL parameters

app.post("/users/:id", (req, res) => res.send(`User ID: ${req.params.id}`));

// Sending custom status code

app.post("/status", (req, res) => res.status(201).send("Created!"));

// Middleware logging

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

console.log("Logging request...");

next();

}, (req, res) => res.send("Logged request!"));

// File upload

app.post("/upload", upload.single("file"), (req, res) => res.send(`File uploaded: ${req.file.originalname}`));

// Handle 404

app.use((req, res) => res.status(404).send("Not Found"));

app.listen(3000, () => console.log("Server running on http://localhost:3000"));

**Summary Table**

| **Feature** | **Method** | **Example** |
| --- | --- | --- |
| **Basic POST** | app.post() | res.send("Hello") |
| **JSON Data** | req.body | req.body.name |
| **Form Data** | req.body | express.urlencoded() |
| **URL Params** | req.params | req.params.id |
| **Headers** | req.headers | req.headers["key"] |
| **Custom Status** | res.status() | res.status(201).send("Created") |
| **Redirect** | res.redirect() | res.redirect("/home") |
| **Middleware** | next() | app.use(logger) |
| **File Upload** | multer | upload.single("file") |

**📌 Key Takeaways**

✅Use req.body for JSON and form data.  
✅Use req.params for URL parameters.  
✅Use req.headers to read HTTP headers.  
✅Use multer for file uploads.  
✅ Always handle unknown routes (404).

**Express.js Routing**

Routing in **Express.js** allows you to define different URLs (endpoints) and specify how your server should respond to **GET, POST, PUT, DELETE**, and other HTTP requests.

**1. Basic Routing in Express.js**

🔹Define routes using app.METHOD(PATH, HANDLER).  
🔹METHOD can be **GET, POST, PUT, DELETE**, etc.  
🔹PATH is the **URL endpoint**.  
🔹 HANDLER is the **function that processes the request**.

**Example: Basic Routes**

js

CopyEdit

const express = require("express");

const app = express();

// Home route

app.get("/", (req, res) => res.send("Welcome to Express!"));

// About route

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

// Contact route

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

// Start the server

app.listen(3000, () => console.log("Server running on http://localhost:3000"));

✅ **URLs:**

* http://localhost:3000/ → "Welcome to Express!"
* http://localhost:3000/about → "About Page"
* http://localhost:3000/contact → "Contact Page"

**2. Route Parameters (req.params)**

🔹 **Dynamic values** in the URL using :paramName.  
🔹 Access parameters using req.params.

**Example: Dynamic Routes**

js

CopyEdit

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

res.send(`User ID: ${req.params.id}`);

});

✅**URL:**http://localhost:3000/user/101  
✅ **Response:** "User ID: 101"

**Example: Multiple Route Parameters**

js

CopyEdit

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

res.send(`Category: ${req.params.category}, Post ID: ${req.params.id}`);

});

✅**URL:**http://localhost:3000/post/tech/500  
✅ **Response:** "Category: tech, Post ID: 500"

**3. Query Parameters (req.query)**

🔹Used for **optional** data, usually after ?.  
🔹Multiple parameters are separated using &.  
🔹 Access query parameters using req.query.

**Example: Handling Query Parameters**

js

CopyEdit

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

const { keyword, page } = req.query;

res.send(`Searching for "${keyword}" on page ${page}`);

});

✅ **URL:**

bash

CopyEdit

http://localhost:3000/search?keyword=NodeJS&page=2

✅ **Response:** "Searching for 'NodeJS' on page 2"

**4. Handling Different HTTP Methods**

🔹 Express supports **GET, POST, PUT, DELETE** for CRUD operations.

**Example: CRUD Routes**

js

CopyEdit

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

app.get("/products", (req, res) => res.send("Fetching all products")); // Read

app.post("/products", (req, res) => res.send("Creating a product")); // Create

app.put("/products/:id", (req, res) => res.send(`Updating product ${req.params.id}`)); // Update

app.delete("/products/:id", (req, res) => res.send(`Deleting product ${req.params.id}`)); // Delete

✅ **Requests:**

* GET /products → "Fetching all products"
* POST /products → "Creating a product"
* PUT /products/5 → "Updating product 5"
* DELETE /products/5 → "Deleting product 5"

**5. Router-Level Middleware**

🔹 Use express.Router() to manage routes **modularly**.

**Example: Using Express Router**

**📁 routes/userRoutes.js**

js

CopyEdit

const express = require("express");

const router = express.Router();

// User routes

router.get("/", (req, res) => res.send("User List"));

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

module.exports = router;

**📁 server.js**

js

CopyEdit

const express = require("express");

const app = express();

const userRoutes = require("./routes/userRoutes");

app.use("/users", userRoutes); // Mount user routes

app.listen(3000, () => console.log("Server running on http://localhost:3000"));

✅ **URLs:**

* http://localhost:3000/users/ → "User List"
* http://localhost:3000/users/5 → "User ID: 5"

**6. Route Chaining for the Same Path**

🔹 Use app.route() to group **multiple HTTP methods** for the same path.

**Example: Route Chaining**

js

CopyEdit

app.route("/item")

.get((req, res) => res.send("Fetching item"))

.post((req, res) => res.send("Creating item"))

.put((req, res) => res.send("Updating item"))

.delete((req, res) => res.send("Deleting item"));

✅ **Requests:**

* GET /item → "Fetching item"
* POST /item → "Creating item"
* PUT /item → "Updating item"
* DELETE /item → "Deleting item"

**7. Handling 404 Errors (Unknown Routes)**

🔹 Use app.use() to catch all **undefined routes**.

**Example: Handling 404**

js

CopyEdit

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

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

});

✅ **Accessing a non-existent route returns:** "Page not found" (HTTP 404).

**8. Redirecting Routes**

🔹 Use res.redirect() to send users to another URL.

**Example: Redirecting to Another Route**

js

CopyEdit

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

res.redirect("/new-route");

});

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

res.send("This is the new route");

});

✅ **URL:** http://localhost:3000/old-route  
✅ **Redirects to:** "This is the new route"

**9. Using Middleware for Routes**

🔹 Middleware functions process requests **before** reaching the route handler.

**Example: Middleware in Routing**

js

CopyEdit

const logger = (req, res, next) => {

console.log(`Request to: ${req.originalUrl}`);

next();

};

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

res.send("Dashboard Page");

});

✅ **Logs URL before sending the response.**

**10. Full Express Routing Example**

js

CopyEdit

const express = require("express");

const app = express();

app.use(express.json());

// Basic routes

app.get("/", (req, res) => res.send("Welcome to Express!"));

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

// Dynamic route with URL parameter

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

// Query parameters

app.get("/search", (req, res) => res.send(`Search: ${req.query.keyword}`));

// CRUD operations

app.route("/products")

.get((req, res) => res.send("Fetching products"))

.post((req, res) => res.send("Creating product"));

app.put("/products/:id", (req, res) => res.send(`Updating product ${req.params.id}`));

app.delete("/products/:id", (req, res) => res.send(`Deleting product ${req.params.id}`));

// Handle unknown routes

app.use((req, res) => res.status(404).send("Not Found"));

app.listen(3000, () => console.log("Server running on http://localhost:3000"));

**Summary Table**

| **Feature** | **Method** | **Example** |
| --- | --- | --- |
| **Basic Routing** | app.get() | app.get("/", handler) |
| **Route Parameters** | req.params | /users/:id |
| **Query Parameters** | req.query | /search?keyword=node |
| **Router Middleware** | express.Router() | router.get("/", handler) |
| **Route Chaining** | app.route() | .get().post().put() |
| **Redirect** | res.redirect() | res.redirect("/new") |
| **404 Handling** | app.use() | res.status(404).send("Not found") |

**📌 Key Takeaways**

✅Use req.params for dynamic URL segments.  
✅Use req.query for optional query parameters.  
✅Use express.Router() for modular routes.  
✅Group HTTP methods using app.route().  
✅ Handle unknown routes with a 404 response.

**Express.js Cookies**

Cookies in **Express.js** are used to **store small pieces of data** in the user's browser, such as authentication tokens or user preferences.

**1. Installing Cookie Parser Middleware**

To work with cookies, install the cookie-parser middleware:

sh

CopyEdit

npm install cookie-parser

**Import & Use Middleware**

js

CopyEdit

const express = require("express");

const cookieParser = require("cookie-parser");

const app = express();

app.use(cookieParser()); // Enables cookie handling

**2. Setting Cookies (res.cookie())**

You can set cookies using the res.cookie(name, value, options) method.

**Example: Setting a Cookie**

js

CopyEdit

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

res.cookie("username", "JohnDoe", { maxAge: 900000, httpOnly: true });

res.send("Cookie has been set!");

});

✅ **Options:**

* maxAge: 900000 → Cookie expires in **15 minutes**.
* httpOnly: true → Cookie is **not accessible** via JavaScript.

✅ **Response:** "Cookie has been set!"

Check in **DevTools → Application → Cookies**.

**3. Reading Cookies (req.cookies)**

Use req.cookies to read cookies.

**Example: Reading a Cookie**

js

CopyEdit

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

res.send(`Cookie value: ${req.cookies.username}`);

});

✅ **URL:** http://localhost:3000/get-cookie  
✅ **Response:** "Cookie value: JohnDoe"

**4. Deleting Cookies (res.clearCookie())**

Use res.clearCookie(name) to delete a cookie.

**Example: Deleting a Cookie**

js

CopyEdit

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

res.clearCookie("username");

res.send("Cookie has been deleted!");

});

✅ **Response:** "Cookie has been deleted!"

**5. Secure Cookies (secure: true)**

🔹 Cookies can be made **secure** by enabling the secure flag, which ensures they are sent **only over HTTPS**.

**Example: Secure Cookie**

js

CopyEdit

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

res.cookie("secureData", "SensitiveInfo", { secure: true, httpOnly: true });

res.send("Secure cookie set!");

});

✅ **Requires HTTPS** (secure: true only works over HTTPS).

**6. Signed Cookies (signed: true)**

🔹 Signed cookies ensure **data integrity** using a secret key.

**Enable Signed Cookies**

js

CopyEdit

app.use(cookieParser("mySecretKey")); // Provide a secret key

**Example: Setting a Signed Cookie**

js

CopyEdit

app.get("/set-signed-cookie", (req, res) => {

res.cookie("authToken", "abc123", { signed: true, httpOnly: true });

res.send("Signed cookie set!");

});

**Example: Reading a Signed Cookie**

js

CopyEdit

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

res.send(`Signed Cookie: ${req.signedCookies.authToken}`);

});

✅**Signed cookies are stored as encrypted values.**  
✅ Prevents users from **modifying cookies** manually.

**7. Full Express.js Cookie Example**

js

CopyEdit

const express = require("express");

const cookieParser = require("cookie-parser");

const app = express();

app.use(cookieParser("mySecretKey")); // Middleware with signing

// Set a cookie

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

res.cookie("username", "JohnDoe", { maxAge: 900000, httpOnly: true });

res.send("Cookie set!");

});

// Read a cookie

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

res.send(`Cookie: ${req.cookies.username}`);

});

// Delete a cookie

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

res.clearCookie("username");

res.send("Cookie deleted!");

});

// Set a signed cookie

app.get("/set-signed-cookie", (req, res) => {

res.cookie("authToken", "abc123", { signed: true, httpOnly: true });

res.send("Signed cookie set!");

});

// Read a signed cookie

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

res.send(`Signed Cookie: ${req.signedCookies.authToken}`);

});

app.listen(3000, () => console.log("Server running on http://localhost:3000"));

**8. Summary Table**

| **Feature** | **Method** | **Example** |
| --- | --- | --- |
| **Set Cookie** | res.cookie() | res.cookie("key", "value") |
| **Read Cookie** | req.cookies | req.cookies.key |
| **Delete Cookie** | res.clearCookie() | res.clearCookie("key") |
| **Secure Cookie** | secure: true | res.cookie("key", "value", { secure: true }) |
| **Signed Cookie** | signed: true | res.cookie("key", "value", { signed: true }) |

**📌 Key Takeaways**

✅**Use cookie-parser middleware** to manage cookies.  
✅**Secure cookies** with httpOnly: true and secure: true.  
✅**Signed cookies** prevent tampering.  
✅ **Clear cookies** using res.clearCookie().

**Express.js File Upload**

Uploading files in **Express.js** is done using middleware like multer, which handles multipart/form-data requests.

**1. Install Multer**

First, install multer:

sh

CopyEdit

npm install multer

**2. Basic File Upload Example**

🔹 Use multer.diskStorage() to configure storage options.  
🔹 Define destination (upload folder) and filename (to customize file names).

js

CopyEdit

const express = require("express");

const multer = require("multer");

const path = require("path");

const app = express();

// Configure Multer Storage

const storage = multer.diskStorage({

destination: "./uploads/", // Uploads will be stored in "uploads" folder

filename: (req, file, cb) => {

cb(null, file.fieldname + "-" + Date.now() + path.extname(file.originalname));

}

});

// Initialize Multer

const upload = multer({ storage });

// Route to handle file upload

app.post("/upload", upload.single("file"), (req, res) => {

res.send(`File uploaded: ${req.file.filename}`);

});

// Start server

app.listen(3000, () => console.log("Server running on http://localhost:3000"));

✅ **Request:**

* Send a POST request to /upload with **multipart/form-data**.
* The file should be attached under the field name **"file"**.

✅ **Response:**

arduino

CopyEdit

File uploaded: file-17123456789.jpg

✅ **Uploaded file will be saved in uploads/ folder**.

**3. Handling Multiple Files**

🔹 Use upload.array() for multiple file uploads.

js

CopyEdit

app.post("/upload-multiple", upload.array("files", 5), (req, res) => {

res.send(`Uploaded ${req.files.length} files successfully!`);

});

✅**Uploads up to 5 files at once.**  
✅ **Files must be sent with the field name "files"**.

**4. File Size & Type Restrictions**

🔹 Use limits to restrict file sizes.  
🔹 Use fileFilter to allow only specific file types.

js

CopyEdit

const uploadFiltered = multer({

storage,

limits: { fileSize: 2 \* 1024 \* 1024 }, // 2MB max file size

fileFilter: (req, file, cb) => {

const allowedTypes = /jpeg|jpg|png|gif/;

const ext = path.extname(file.originalname).toLowerCase();

if (allowedTypes.test(ext)) {

cb(null, true);

} else {

cb(new Error("Only images are allowed!"));

}

}

});

app.post("/upload-image", uploadFiltered.single("image"), (req, res) => {

res.send(`Image uploaded: ${req.file.filename}`);

});

✅**Allows only .jpeg, .jpg, .png, .gif files**.  
✅**Restricts file size to 2MB**.  
✅ **Returns an error for unsupported file types**.

**5. Storing Files in Memory (Buffer)**

🔹 Useful when processing files **before saving** to cloud storage (AWS S3, Firebase, etc.).

js

CopyEdit

const uploadMemory = multer({ storage: multer.memoryStorage() });

app.post("/upload-buffer", uploadMemory.single("file"), (req, res) => {

res.send(`File received in buffer: ${req.file.originalname}`);

});

✅**File is stored in req.file.buffer instead of disk.**  
✅ **Good for cloud uploads (AWS, Firebase, etc.).**

**6. Complete File Upload Example**

js

CopyEdit

const express = require("express");

const multer = require("multer");

const path = require("path");

const app = express();

app.use(express.static("public")); // Serve static files (optional)

// Multer Storage Config

const storage = multer.diskStorage({

destination: "./uploads/",

filename: (req, file, cb) => {

cb(null, file.fieldname + "-" + Date.now() + path.extname(file.originalname));

}

});

const upload = multer({ storage });

// Routes

app.post("/upload", upload.single("file"), (req, res) => {

res.send(`File uploaded: ${req.file.filename}`);

});

app.post("/upload-multiple", upload.array("files", 5), (req, res) => {

res.send(`Uploaded ${req.files.length} files`);

});

// Start Server

app.listen(3000, () => console.log("Server running on http://localhost:3000"));

**7. Frontend HTML for Testing**

Create index.html in the public/ folder:

html

CopyEdit

<!DOCTYPE html>

<html lang="en">

<head>

<title>File Upload</title>

</head>

<body>

<h2>Upload a File</h2>

<form action="/upload" method="post" enctype="multipart/form-data">

<input type="file" name="file">

<button type="submit">Upload</button>

</form>

<h2>Upload Multiple Files</h2>

<form action="/upload-multiple" method="post" enctype="multipart/form-data">

<input type="file" name="files" multiple>

<button type="submit">Upload</button>

</form>

</body>

</html>

✅ **Test the API with this simple HTML form.**

**8. Summary Table**

| **Feature** | **Multer Method** | **Example** |
| --- | --- | --- |
| **Single File Upload** | upload.single("file") | /upload |
| **Multiple Files** | upload.array("files", 5) | /upload-multiple |
| **File Size Limit** | limits: { fileSize: 2 \* 1024 \* 1024 } | Restricts file size |
| **Restrict File Types** | fileFilter: (req, file, cb) => {...} | Allows only images |
| **Store in Memory** | storage: multer.memoryStorage() | req.file.buffer |

**📌 Key Takeaways**

✅Use multer to handle file uploads in Express.js.  
✅Store files on disk (multer.diskStorage()) or memory (multer.memoryStorage()).  
✅Set **file size & type restrictions** for security.  
✅ Upload **single or multiple** files easily.

**Express.js Middleware**

Middleware in **Express.js** are functions that execute **before** reaching the final request handler. They can **modify requests, responses, and control flow** in the application.

**1. What is Middleware?**

Middleware functions:

* Execute **before** the request reaches the route handler.
* Have access to req, res, and next().
* Can **modify** the request (req), response (res), or pass control using next().

**Basic Middleware Example**

js

CopyEdit

const express = require("express");

const app = express();

// Custom Middleware

const logger = (req, res, next) => {

console.log(`Request: ${req.method} ${req.url}`);

next(); // Pass control to the next middleware

};

app.use(logger); // Use middleware globally

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

res.send("Hello, Middleware!");

});

app.listen(3000, () => console.log("Server running on http://localhost:3000"));

✅ **Logs every request** before reaching the route.

**2. Types of Middleware**

Express has different types of middleware:

| **Type** | **Example** | **Description** |
| --- | --- | --- |
| **Application-Level** | app.use(logger) | Runs for all routes. |
| **Route-Specific** | app.get("/", middleware, handler) | Runs only for specific routes. |
| **Built-in** | express.json() | Handles JSON request bodies. |
| **Third-Party** | cookie-parser | External libraries like multer, cors. |
| **Error Handling** | app.use((err, req, res, next) => {...}) | Handles errors globally. |

**3. Built-in Middleware**

Express has built-in middleware for common tasks.

**📌 3.1 express.json() – Parse JSON Body**

Parses application/json requests.

js

CopyEdit

app.use(express.json()); // Enable JSON parsing

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

res.send(`Received: ${JSON.stringify(req.body)}`);

});

✅ **Required for handling JSON payloads in POST requests.**

**📌 3.2 express.urlencoded() – Parse Form Data**

Parses application/x-www-form-urlencoded (form submissions).

js

CopyEdit

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

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

res.send(`Form Data: ${req.body.name}`);

});

✅ **Use this for handling HTML form submissions.**

**📌 3.3 express.static() – Serve Static Files**

Serves static files like CSS, JS, images.

js

CopyEdit

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

✅ **Now public/index.html can be accessed directly.**

**4. Third-Party Middleware**

Third-party middleware extends Express functionality.

**📌 4.1 cors – Enable Cross-Origin Requests**

Used to allow API requests from different domains.

sh

CopyEdit

npm install cors

js

CopyEdit

const cors = require("cors");

app.use(cors()); // Enable CORS

✅ **Required for cross-domain API calls.**

**📌 4.2 morgan – Logging Middleware**

Logs all HTTP requests.

sh

CopyEdit

npm install morgan

js

CopyEdit

const morgan = require("morgan");

app.use(morgan("dev"));

✅ **Logs each request with details like method, status, and response time.**

**📌 4.3 cookie-parser – Handling Cookies**

Parses cookies from requests.

sh

CopyEdit

npm install cookie-parser

js

CopyEdit

const cookieParser = require("cookie-parser");

app.use(cookieParser());

✅ **Allows reading cookies using req.cookies.**

**5. Route-Specific Middleware**

Middleware can be applied to specific routes.

js

CopyEdit

const authMiddleware = (req, res, next) => {

if (req.query.token === "12345") {

next(); // Continue if token is correct

} else {

res.status(403).send("Forbidden");

}

};

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

res.send("Access granted!");

});

✅ **Only runs for /secure route.**

**6. Error Handling Middleware**

Error-handling middleware is defined **after all routes**.

js

CopyEdit

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

console.error(err.stack);

res.status(500).send("Something went wrong!");

});

✅ **Handles errors globally.**

**7. Complete Middleware Example**

js

CopyEdit

const express = require("express");

const morgan = require("morgan");

const cors = require("cors");

const app = express();

app.use(express.json()); // Parse JSON

app.use(express.urlencoded({ extended: true })); // Parse form data

app.use(express.static("public")); // Serve static files

app.use(cors()); // Enable CORS

app.use(morgan("dev")); // Log requests

// Custom Middleware

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

console.log(`Request received: ${req.method} ${req.url}`);

next();

});

// Routes

app.get("/", (req, res) => res.send("Hello Middleware!"));

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

throw new Error("Intentional Error");

});

// Error Handling Middleware

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

console.error(err.stack);

res.status(500).send("Internal Server Error");

});

// Start Server

app.listen(3000, () => console.log("Server running on http://localhost:3000"));

**8. Summary Table**

| **Middleware Type** | **Example** | **Use Case** |
| --- | --- | --- |
| **Application-Level** | app.use(middleware) | Runs for all requests |
| **Route-Specific** | app.get("/", middleware, handler) | Runs only on specific routes |
| **Built-in Middleware** | express.json(), express.static() | Handles JSON, static files, etc. |
| **Third-Party Middleware** | morgan, cors, cookie-parser | Additional features |
| **Error Handling** | app.use((err, req, res, next) => {...}) | Catches and handles errors |

**📌 Key Takeaways**

✅Middleware runs **before** route handlers.  
✅Use app.use() for **global middleware**.  
✅Use **third-party middleware** for logging, security, etc.  
✅ **Error-handling middleware** catches and logs errors.

**Express.js Scaffolding**

Scaffolding in **Express.js** means generating a structured project layout **automatically** instead of manually creating folders and files. This is done using the express-generator tool.

**1. Install Express Generator**

First, install express-generator globally:

sh

CopyEdit

npm install -g express-generator

Now you can create a new Express project with a predefined folder structure.

**2. Generate an Express App**

To scaffold a new Express app, run:

sh

CopyEdit

express myApp

or with **EJS templating**:

sh

CopyEdit

express --view=ejs myApp

This creates a folder named myApp with a structured Express project.

**3. Project Structure**

After running the command, the generated project will look like this:

php

CopyEdit

myApp/

│── bin/ # Start script

│── public/ # Static assets (CSS, images, JS)

│── routes/ # Route files

│ ├── index.js # Main routes

│ ├── users.js # User-related routes

│── views/ # Template files (if using EJS, Pug, etc.)

│── app.js # Main Express app file

│── package.json # Project metadata & dependencies

**4. Install Dependencies**

Navigate into your project folder:

sh

CopyEdit

cd myApp

npm install

Then start the server:

sh

CopyEdit

npm start

**5. Understanding the Files**

| **File/Folder** | **Purpose** |
| --- | --- |
| **app.js** | Main Express application file |
| **bin/www** | Starts the server |
| **public/** | Stores static files (CSS, JS, images) |
| **routes/** | Defines application routes |
| **views/** | Stores template files (if using EJS/Pug) |

**6. Define Custom Routes**

Modify routes/index.js:

js

CopyEdit

const express = require("express");

const router = express.Router();

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

res.send("Welcome to Express Scaffolding!");

});

module.exports = router;

**7. Running the Project**

Start the server:

sh

CopyEdit

npm start

Now, open **http://localhost:3000/** in your browser. 🎉

**📌 Key Takeaways**

✅express-generator quickly scaffolds an Express.js project.  
✅It creates a **structured** directory with routes, views, and public.  
✅ npm start launches the server instantly.

**Express.js Templates**

Express.js supports **templating engines** to dynamically render HTML pages. Some popular template engines are:

* **EJS (Embedded JavaScript)**
* **Pug (formerly Jade)**
* **Handlebars (hbs)**

**1. Setting Up a Template Engine**

You need to specify a view engine in your Express app.

**Example: Using EJS**

1️⃣ Install EJS:

sh

CopyEdit

npm install ejs

2️⃣ Set up the view engine in app.js:

js

CopyEdit

const express = require("express");

const app = express();

// Set view engine to EJS

app.set("view engine", "ejs");

// Define a route

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

res.render("index", { title: "Home Page", message: "Welcome to Express!" });

});

// Start the server

app.listen(3000, () => console.log("Server running on http://localhost:3000"));

3️⃣ Create the views/index.ejs file:

ejs

CopyEdit

<!DOCTYPE html>

<html lang="en">

<head>

<title><%= title %></title>

</head>

<body>

<h1><%= message %></h1>

</body>

</html>

✅ Now, running npm start will render this **dynamic HTML page** at http://localhost:3000/.

**2. Using Pug (Alternative to EJS)**

1️⃣ Install Pug:

sh

CopyEdit

npm install pug

2️⃣ Set Pug as the template engine:

js

CopyEdit

app.set("view engine", "pug");

3️⃣ Create views/index.pug:

pug

CopyEdit

doctype html

html

head

title= title

body

h1= message

✅ Pug uses **indentation** instead of HTML tags.

**3. Using Handlebars (HBS)**

1️⃣ Install Handlebars:

sh

CopyEdit

npm install hbs

2️⃣ Set HBS as the template engine:

js

CopyEdit

app.set("view engine", "hbs");

3️⃣ Create views/index.hbs:

hbs

CopyEdit

<!DOCTYPE html>

<html lang="en">

<head>

<title>{{title}}</title>

</head>

<body>

<h1>{{message}}</h1>

</body>

</html>

**4. Template Engine Comparison**

| **Engine** | **Syntax** | **Features** |
| --- | --- | --- |
| **EJS** | <%= variable %> | Similar to HTML, easy to learn |
| **Pug** | title= title | Minimal syntax, indentation-based |
| **Handlebars** | {{ variable }} | Simple and logic-less |

**📌 Key Takeaways**

✅**Express supports multiple template engines**.  
✅**EJS is the easiest to start with** (uses regular HTML syntax).  
✅**Pug has a cleaner syntax but requires indentation**.  
✅ **Handlebars (HBS) is good for simple and readable templates**.

**Differences from Traditional Server-Side Technologies:**

Language: Traditionally, server-side development was dominated by languages like Java, C#, and PHP. Node.js introduced JavaScript as a server-side language, leveraging its popularity and familiarity among frontend developers.

Concurrency Model: Unlike traditional server-side technologies, which often use multi threading or blocking I/O, Node.js employs a single-threaded, event-driven model with non blocking I/O. This makes it more lightweight and efficient in handling concurrent connections and I/O-bound tasks.

Performance: Node.js is known for its high performance and scalability, particularly in handling I/O-bound tasks and real-time applications. Its event-driven architecture allows for efficient resource utilization and low latency responses compared to traditional server-side technologies.

Development Paradigm: Node.js promotes a more asynchronous and callback-based programming style compared to traditional synchronous, blocking I/O approaches. This can lead to a different mindset and programming patterns for developers transitioning from traditional server-side technologies to Node.js.

**Overview of the Event Loop:**

Single-Threaded Model: Node.js operates on a single-threaded event loop model. Unlike traditional server-side technologies that typically use multi-threading to handle concurrent connections, Node.js uses a single thread to handle all I/O operations and execute JavaScript code.

Event-Driven Architecture: Node.js is event-driven, meaning it uses events to trigger the execution of asynchronous operations. Events can be triggered by various sources, such as incoming HTTP requests, file system operations, timers, or external APIs.

Non-Blocking I/O: Node.js uses non-blocking, asynchronous I/O operations. When Node.js performs an I/O operation, such as reading from a file or making an HTTP request, it doesn't wait for the operation to complete. Instead, it continues executing other code while monitoring the status of the I/O operation.

Callback Functions: When an asynchronous operation is initiated, Node.js provides a callback function to be executed once the operation is completed or an event is triggered. This callback function is placed in an event queue to be processed by the event loop.

**Event Loop Phases:**

Poll Phase: In the poll phase, the event loop retrieves events from the event queue and executes their associated callback functions synchronously. If there are no events in the queue, the event loop waits for new events to arrive.

Timers Phase: The event loop checks for any scheduled timer events whose time has expired. If there are any, it executes their callback functions.

Pending Callbacks Phase: In this phase, the event loop executes I/O callback functions that were deferred during the poll phase. These callback functions are typically triggered when asynchronous I/O operations complete or when events are emitted.

Idle, Prepare Phases: These phases are internal to the event loop and are mainly used for internal bookkeeping. Close Handlers Phase: In the close handlers phase, the event loop executes any close event handlers, such as closing database connections or server instances.

**Efficiency in Handling Asynchronous Operations:**

Non-Blocking Nature: Node.js' event loop allows it to perform asynchronous operations without blocking the execution of other code. This enables Node.js to handle multiple concurrent connections efficiently, making it suitable for real-time applications and high throughput systems.

Callback-Based Approach: By using callback functions, Node.js can continue executing other code while waiting for asynchronous operations to complete. This callback-based approach allows developers to write code that is more readable and maintainable compared to traditional blocking I/O approaches.

Scalability: The event loop model contributes to Node.js' scalability by allowing it to handle a large number of concurrent connections with minimal resources. This scalability is essential for building high-performance, scalable applications and services.

**1. First class function in Javascript?**

When functions can be treated like any other variable then those functions are first-class functions. There are many other programming languages, for example, scala, Haskell, etc which follow this including JS. Now because of this function can be passed as a param to another function(callback) or a function can return another function(higher-order function). map() and filter() are higher-order functions that are popularly used.

### 2. Node.js and how it works?

Node.js is a virtual machine that uses JavaScript as its scripting language and runs Chrome’s V8 JavaScript engine. Basically, Node.js is based on an event-driven architecture where I/O runs asynchronously making it lightweight and efficient. It is being used in developing desktop applications as well with a popular framework called electron as it provides API to access OS-level features such as file system, network, etc.

### 3. Manage packages in your node.js project?

It can be managed by a number of package installers and their configuration file accordingly. Out of them mostly use npm or yarn. Both provide almost all libraries of javascript with extended features of controlling environment-specific configurations. To maintain versions of libs being installed in a project we use package.json and package-lock.json so that there is no issue in porting that app to a different environment.

**Concept of middleware in Express.js. with examples of commonly used middleware function.**

Answer/Solution: Middleware in Express.js are functions that have access to the request object (req), the response object (res), and the next middleware function in the application's request response cycle.

Middleware functions can perform tasks such as modifying the request and response objects, executing code, terminating the request-response cycle, and calling the next middleware function in the stack.

**Middleware functions in Express.js can be used for various purposes, including:**

Logging: Logging middleware can be used to log information about incoming requests, such as the request method, URL, and timestamp. This information can be useful for debugging, monitoring, and auditing purposes.

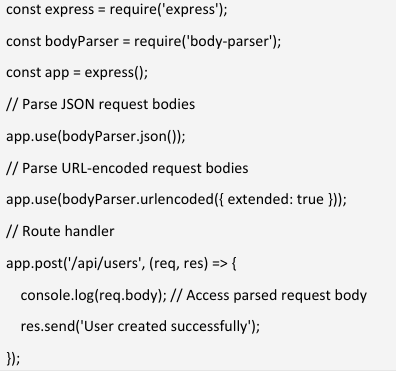
Authentication and Authorization: Middleware functions can be used to authenticate and authorize users before allowing them to access protected routes or resources. Authentication middleware can verify the user's identity, while authorization middleware can check whether the user has the necessary permissions to perform the requested action.

Error Handling: Error handling middleware can catch errors that occur during the request response cycle and handle them appropriately. This can include logging the error, sending an error response to the client, or performing cleanup tasks.

Parsing Request Bodies: Middleware functions can parse incoming request bodies, such as JSON, URL-encoded, or multipart form data, and make the parsed data available in the request object for further processing by route handlers.

Compression: Compression middleware can compress outgoing responses to reduce the size of data transferred over the network. This can improve the performance of web applications by reducing bandwidth usage and decreasing page load times.

Ex(i) Body Parser Middleware: The body-parser middleware is used to parse incoming request bodies before they are handled by route handlers. It can parse various types of request bodies, including JSON, URL-encoded, and multipart form data, and make the parsed data available in the req.body property.



Ex(ii) Logger Middleware: A custom logger middleware can be used to log information about incoming requests. This middleware logs the request method, URL, and timestamp to the console before passing the request to the next middleware function.

Custom logger middleware

function logger(req, res, next) {

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

next(); // Call the next middleware function

}

// Register logger middleware

app.use(logger);

### 4. How is Node.js better than other frameworks most popularly used?

* Node.js provides simplicity in development because of its non-blocking I/O and event-based model results in short response time and concurrent processing, unlike other frameworks where developers have to use thread management.
* It runs on a chrome v8 engine which is written in c++ and is highly performant with constant improvement.
* Also since we will use Javascript in both the frontend and backend the development will be much faster.
* And at last, there are sample libraries so that we don’t need to reinvent the wheel.

**6. What are some commonly used timing features of Node.js?**

* **setTimeout/clearTimeout** – This is used to implement delays in code execution.
* **setInterval/clearInterval** – This is used to run a code block multiple times.
* **setImmediate/clearImmediate** – Any function passed as the setImmediate() argument is a callback that's executed in the next iterationof the event loop.
* **process.nextTick** – Both setImmediate and process.nextTick appear to be doing the same thing; however, you may prefer one over the other depending on your callback’s urgency.

### 7. What are the advantages of using promises instead of callbacks?

The main advantage of using promise is you get an object to decide the action that needs to be taken after the async task completes. This gives more manageable code and avoids callback hell.

### 8. What is fork in node JS?

A fork in general is used to spawn child processes. In node it is used to create a new instance of v8 engine to run multiple workers to execute the code.

**# The process of handling HTTP requests and responses using Express.js.**

Client Sends a Request: A client, such as a web browser, sends an HTTP request to the server. The request includes information like the URL, HTTP method (GET, POST, etc.), headers, and optionally, a body with data.

Express.js Routes Handling: Express.js defines routes to handle different types of requests. These routes are defined using HTTP methods and URL patterns. For example:

### 9. Why is Node.js single-threaded?

Node.js was created explicitly as an experiment in async processing. This was to try a new theory of doing async processing on a single thread over the existing thread-based implementation of scaling via different frameworks.

### 10. How do you create a simple server in Node.js that returns Hello World?

var http = require("http");

http.createServer(function (request, response) {

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

response.end('Hello World\n');

}).listen(3000);

**11. How many types of API functions are there in Node.js?**

There are two types of API functions:

* **Asynchronous, non-blocking functions** - mostly I/O operations which can be fork out of the main loop.
* **Synchronous, blocking functions** - mostly operations that influence the process running in the main loop.

### 12. What is REPL?

PL in Node.js stands for **R**ead, **E**val, **P**rint, and **L**oop, which further means evaluating code on the go.

### 13. List down the two arguments that async.queue takes as input?

* Task Function
* Concurrency Value

### 14. What is the purpose of module.exports?

This is used to expose functions of a particular module or file to be used elsewhere in the project. This can be used to encapsulate all similar functions in a file which further improves the project structure.  
  
For example, you have a file for all utils functions with util to get solutions in a different programming language of a problem statement.

const getSolutionInJavaScript = async ({

problem\_id

}) => {

...

};

const getSolutionInPython = async ({

problem\_id

}) => {

...

};

module.exports = { getSolutionInJavaScript, getSolutionInPython }

Thus using module.exports we can use these functions in some other file:

const { getSolutionInJavaScript, getSolutionInPython} = require("./utils")

### 15. What tools can be used to assure consistent code style?

ESLint can be used with any IDE to ensure a consistent coding style which further helps in maintaining the codebase.

### 1. What do you understand by callback hell?

async\_A(function(){

async\_B(function(){

async\_C(function(){

async\_D(function(){

....

});

});

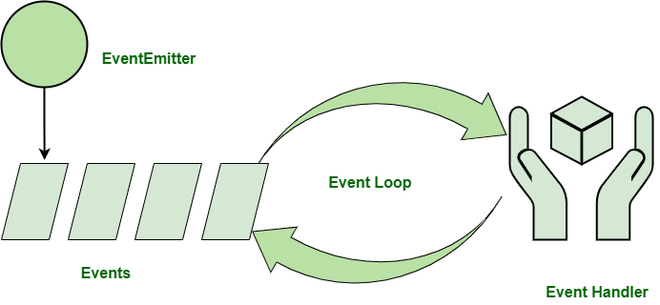
});

});

For the above example, we are passing callback functions and it makes the code unreadable and not maintainable, thus we should change the async logic to avoid this.

### 2. What is an event-loop in Node JS?

Whatever that is async is managed by event-loop using a queue and listener.  We can get the idea using the following diagram:

Node.js Event Loop

So when an async function needs to be executed(or I/O) the main thread sends it to a different thread allowing v8 to keep executing the main code. Event loop involves different phases with specific tasks such as timers, pending callbacks, idle or prepare, poll, check, close callbacks with different FIFO queues. Also in between iterations it checks for async I/O or timers and shuts down cleanly if there aren't any.

### 3. If Node.js is single threaded then how does it handle concurrency?

The main loop is single-threaded and all async calls are managed by libuv library.

For example:

const crypto = require("crypto");

const start = Date.now();

function logHashTime() {

crypto.pbkdf2("a", "b", 100000, 512, "sha512", () => {

console.log("Hash: ", Date.now() - start);

});

}

logHashTime();

logHashTime();

logHashTime();

logHashTime();

This gives the output:

Hash: 1213

Hash: 1225

Hash: 1212

Hash: 1222

This is because libuv sets up a thread pool to handle such concurrency. How many threads will be there in the thread pool depends upon the number of cores but you can override this.

### 4. Differentiate between process.nextTick() and setImmediate()?

Both can be used to switch to an asynchronous mode of operation by listener functions.   
  
process.nextTick() sets the callback to execute but setImmediate pushes the callback in the queue to be executed. So the event loop runs in the following manner  
  
**timers–>pending callbacks–>idle,prepare–>connections(poll,data,etc)–>check–>close callbacks**

In this process.nextTick() method adds the callback function to the start of the next event queue and setImmediate() method to place the function in the check phase of the next event queue.

### 5. How does Node.js overcome the problem of blocking of I/O operations?

Since the node has an event loop that can be used to handle all the I/O operations in an asynchronous manner without blocking the main function.   
  
So for example, if some network call needs to happen it will be scheduled in the event loop instead of the main thread(single thread). And if there are multiple such I/O calls each one will be queued accordingly to be executed separately(other than the main thread).

Thus even though we have single-threaded JS, I/O ops are handled in a nonblocking way.

### 6. How can we use async await in node.js?

Here is an example of using async-await pattern:

// this code is to retry with exponential backoff

function wait (timeout) {

return new Promise((resolve) => {

setTimeout(() => {

resolve()

}, timeout);

});

}

async function requestWithRetry (url) {

const MAX\_RETRIES = 10;

for (let i = 0; i <= MAX\_RETRIES; i++) {

try {

return await request(url);

} catch (err) {

const timeout = Math.pow(2, i);

console.log('Waiting', timeout, 'ms');

await wait(timeout);

console.log('Retrying', err.message, i);

}

}

}

### 7. What is node.js streams?

Streams are instances of EventEmitter which can be used to work with streaming data in Node.js. They can be used for handling and manipulating streaming large files(videos, mp3, etc) over the network. They use buffers as their temporary storage.  
  
There are mainly four types of the stream:

* **Writable:** streams to which data can be written (for example, fs.createWriteStream()).
* **Readable:** streams from which data can be read (for example, fs.createReadStream()).
* **Duplex:** streams that are both Readable and Writable (for example, net.Socket).
* **Transform:** Duplex streams that can modify or transform the data as it is written and read (for example, zlib.createDeflate()).

### 8. What are node.js buffers?

In general, buffers is a temporary memory that is mainly used by stream to hold on to some data until consumed. Buffers are introduced with additional use cases than JavaScript’s Unit8Array and are mainly used to represent a fixed-length sequence of bytes. This also supports legacy encodings like ASCII, utf-8, etc. It is a fixed(non-resizable) allocated memory outside the v8.

### 9. What is middleware?

Middleware comes in between your request and business logic. It is mainly used to capture logs and enable rate limit, routing, authentication, basically whatever that is not a part of business logic. There are third-party middleware also such as body-parser and you can write your own middleware for a specific use case.

### 10. Explain what a Reactor Pattern in Node.js?

Reactor pattern again a pattern for nonblocking I/O operations. But in general, this is used in any event-driven architecture.   
  
There are two components in this: 1. Reactor 2. Handler.  
  
**Reactor**: Its job is to dispatch the I/O event to appropriate handlers  
**Handler**: Its job is to actually work on those events

### 11. Why should you separate Express app and server?

The server is responsible for initializing the routes, middleware, and other application logic whereas the app has all the business logic which will be served by the routes initiated by the server. This ensures that the business logic is encapsulated and decoupled from the application logic which makes the project more readable and maintainable.

### 12. For Node.js, why Google uses V8 engine?

Well, are there any other options available? Yes, of course, we have [Spidermonkey](https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey) from Firefox, Chakra from Edge but Google’s v8 is the most evolved(since it’s open-source so there’s a huge community helping in developing features and fixing bugs) and fastest(since it’s written in c++) we got till now as a JavaScript and WebAssembly engine. And it is portable to almost every machine known.

### 13. Describe the exit codes of Node.js?

Exit codes give us an idea of how a process got terminated/the reason behind termination.   
  
A few of them are:

* Uncaught fatal exception - (code - 1) - There has been an exception that is not handled
* Unused - (code - 2) - This is reserved by bash
* Fatal Error - (code - 5) - There has been an error in V8 with stderr output of the description
* Internal Exception handler Run-time failure - (code - 7) - There has been an exception when bootstrapping function was called
* Internal JavaScript Evaluation Failure - (code - 4) - There has been an exception when the bootstrapping process failed to return function value when evaluated.

### 14. Explain the concept of stub in Node.js?

Stubs are used in writing tests which are an important part of development. It replaces the whole function which is getting tested.    
  
This helps in scenarios where we need to test:

* External calls which make tests slow and difficult to write (e.g HTTP calls/ DB calls)
* Triggering different outcomes for a piece of code (e.g. what happens if an error is thrown/ if it passes)

For example, this is the function:

const request = require('request');

const getPhotosByAlbumId = (id) => {

const requestUrl = `https://jsonplaceholder.typicode.com/albums/${id}/photos?\_limit=3`;

return new Promise((resolve, reject) => {

request.get(requestUrl, (err, res, body) => {

if (err) {

return reject(err);

}

resolve(JSON.parse(body));

});

});

};

module.exports = getPhotosByAlbumId;

To test this function this is the stub

const expect = require('chai').expect;

const request = require('request');

const sinon = require('sinon');

const getPhotosByAlbumId = require('./index');

describe('with Stub: getPhotosByAlbumId', () => {

before(() => {

sinon.stub(request, 'get')

.yields(null, null, JSON.stringify([

{

"albumId": 1,

"id": 1,

"title": "A real photo 1",

"url": "https://via.placeholder.com/600/92c952",

"thumbnailUrl": "https://via.placeholder.com/150/92c952"

},

{

"albumId": 1,

"id": 2,

"title": "A real photo 2",

"url": "https://via.placeholder.com/600/771796",

"thumbnailUrl": "https://via.placeholder.com/150/771796"

},

{

"albumId": 1,

"id": 3,

"title": "A real photo 3",

"url": "https://via.placeholder.com/600/24f355",

"thumbnailUrl": "https://via.placeholder.com/150/24f355"

}

]));

});

after(() => {

request.get.restore();

});

it('should getPhotosByAlbumId', (done) => {

getPhotosByAlbumId(1).then((photos) => {

expect(photos.length).to.equal(3);

photos.forEach(photo => {

expect(photo).to.have.property('id');

expect(photo).to.have.property('title');

expect(photo).to.have.property('url');

});

done();

});

});

});

**1. What is an Event Emitter in Node.js?**

EventEmitter is a Node.js class that includes all the objects that are basically capable of emitting events. This can be done by attaching named events that are emitted by the object using an eventEmitter.on() function. Thus whenever this object throws an even the attached functions are invoked synchronously.

const EventEmitter = require('events');

class MyEmitter extends EventEmitter {}

const myEmitter = new MyEmitter();

myEmitter.on('event', () => {

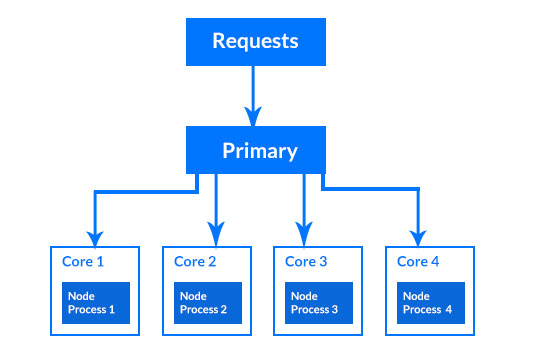
console.log('an event occurred!');

});

myEmitter.emit('event');

**2. Enhancing Node.js performance through clustering.**

Node.js applications run on a single processor, which means that by default they don’t take advantage of a multiple-core system. Cluster mode is used to start up multiple node.js processes thereby having multiple instances of the event loop. When we start using cluster in a nodejs app behind the scene multiple node.js processes are created but there is also a parent process called the **cluster manager** which is responsible for monitoring the health of the individual instances of our application.

Clustering in Node.js

**3. What is a thread pool and which library handles it in Node.js**

The Thread pool is handled by the libuv library. libuv is a multi-platform C library that provides support for asynchronous I/O-based operations such as file systems, networking, and concurrency.

Thread Pool

**4. What is WASI and why is it being introduced?**

Web assembly provides an implementation of [WebAssembly System Interface](https://wasi.dev/) specification through WASI API in node.js implemented using WASI class. The introduction of WASI was done by keeping in mind its possible to use the underlying operating system via a collection of POSIX-like functions thus further enabling the application to use resources more efficiently and features that require system-level access.

**5. How are worker threads different from clusters?**

**Cluster:**

* There is one process on each CPU with an IPC to communicate.
* In case we want to have multiple servers accepting HTTP requests via a single port, clusters can be helpful.
* The processes are spawned in each CPU thus will have separate memory and node instance which further will lead to memory issues.

**Worker threads:**

* There is only one process in total with multiple threads.
* Each thread has one Node instance (one event loop, one JS engine) with most of the APIs accessible.
* Shares memory with other threads (e.g. SharedArrayBuffer)
* This can be used for CPU-intensive tasks like processing data or accessing the file system since NodeJS is single-threaded, synchronous tasks can be made more efficient leveraging the worker's threads.

**6. How to measure the duration of async operations?**

Performance API provides us with tools to figure out the necessary performance metrics. A simple example would be using async\_hooks and perf\_hooks

'use strict';

const async\_hooks = require('async\_hooks');

const {

performance,

PerformanceObserver

} = require('perf\_hooks');

const set = new Set();

const hook = async\_hooks.createHook({

init(id, type) {

if (type === 'Timeout') {

performance.mark(`Timeout-${id}-Init`);

set.add(id);

}

},

destroy(id) {

if (set.has(id)) {

set.delete(id);

performance.mark(`Timeout-${id}-Destroy`);

performance.measure(`Timeout-${id}`,

`Timeout-${id}-Init`,

`Timeout-${id}-Destroy`);

}

}

});

hook.enable();

const obs = new PerformanceObserver((list, observer) => {

console.log(list.getEntries()[0]);

performance.clearMarks();

observer.disconnect();

});

obs.observe({ entryTypes: ['measure'], buffered: true });

setTimeout(() => {}, 1000);

This would give us the exact time it took to execute the callback.

**7. How to measure the performance of async operations?**

Performance API provides us with tools to figure out the necessary performance metrics.   
  
A simple example would be:

const { PerformanceObserver, performance } = require('perf\_hooks');

const obs = new PerformanceObserver((items) => {

console.log(items.getEntries()[0].duration);

performance.clearMarks();

});

obs.observe({ entryTypes: ['measure'] });

performance.measure('Start to Now');

performance.mark('A');

doSomeLongRunningProcess(() => {

performance.measure('A to Now', 'A');

performance.mark('B');

performance.measure('A to B', 'A', 'B');

});