UNIT- IV: NODE JS

Node.js Overview

Node.js is a **runtime environment** that allows you to run JavaScript **on the server-side**, outside of a browser. It's built on **Chrome's V8 JavaScript engine**, which makes it **fast and efficient** for building scalable and high-performance network applications.

What is Node.js?

- **Open-source**, cross-platform
- Executes JavaScript code **outside** a web browser
- Designed for **non-blocking**, event-driven servers
- Ideal for **data-intensive real-time applications** (like chat apps)

Key Features

Feature	Description
Asynchronous and Event-Driven	All APIs are non-blocking; Node.js handles multiple requests simultaneously.
Fast Execution	Built on Google Chrome's V8 engine, which compiles JavaScript to native machine code.
Single Programming Language	Use JavaScript both on the **frontend** and **backend**.
Package Ecosystem	Comes with **npm** (Node Package Manager) – the largest ecosystem of open-source libraries.
Cross-platform	Runs on Windows, macOS, and Linux.

Node.js Architecture

- **Single-threaded** with **event loop**
- Uses **non-blocking I/O** operations (vs blocking I/O in traditional systems)
- Can handle **thousands of concurrent connections** efficiently

UNIT- IV: NODE JS

Common Use Cases

- Real-time applications (e.g. **chat apps**, live updates)
- REST APIs and GraphQL servers
- Microservices
- Streaming services
- Command-line tools
- IoT applications

Example: Simple Node.js Server

```
// Load HTTP module
const http = require('http');

// Create HTTP server
const server = http.createServer((req, res) => {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello from Node.js!\n');
});

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

npm – Node Package Manager

- Use npm to install third-party packages
- Example: npm install express installs Express.js (a popular Node.js web framework)

Popular Node.js Frameworks

- **Express.js** – Minimalist and flexible web framework
- **NestJS** – Scalable and structured, uses TypeScript
- **Fastify** – High-performance HTTP framework
- **Socket.io** – Real-time bidirectional event-based communication

UNIT- IV: NODE JS

Why Choose Node.js?

- Great for **real-time** apps (chat, collaboration tools)
- Easy to learn if you already know JavaScript
- Huge developer community
- Scalable for modern application

Node.js Basics and Setup

1. What You Need to Get Started

- A computer (Windows, macOS, Linux)
- Basic knowledge of JavaScript
- Internet connection to download Node.js

2. Installing Node.js

1. Go to the official Node.js website:
   https://nodejs.org/
2. You will see two versions:
   - **LTS (Long Term Support)** — recommended for most users and production
   - **Current** — latest features but less stable
3. Download the **LTS version** for your OS and install it using the installer.
4. After installation, verify by opening your terminal (Command Prompt, PowerShell, or Terminal) and typing:

```
node -v
npm -v
```

You should see the versions printed out, confirming the installation.

3. Your First Node.js Script

Create a file called app.js anywhere on your computer with the following content:

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

Run this script in the terminal:

node app.js

You should see the output:

Hello, Node.js!

4. Running a Simple HTTP Server

Create a file called server.js with:

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

const server = http.createServer((req, res) => {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello from Node.js server!');
});

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

Run the server:

node server.js

Open your browser and go to http://localhost:3000 to see the message.

5. Using npm (Node Package Manager)

- Initialize a new Node.js project:

npm init

This will prompt you to enter details about your project and create a package.json file.

- To install a package (example: Express.js):

npm install express

- The installed package goes into the node_modules folder and is recorded in package.json.

6. Basic Project Structure Example

my-node-app/
├── app.js
├── package.json
├── node_modules/

- app.js — Your main script
- package.json — Project metadata and dependencies
- node_modules/ — Installed packages

7. Helpful Commands

Command	Purpose
node <filename>	Run a Node.js file
npm init	Initialize a new Node.js project
npm install <name>	Install a package
npm start	Run the start script (defined in package.json)
npm uninstall <name>	Remove a package

Summary:

- Install Node.js and npm
- Run JavaScript files with node
- Create simple servers with built-in HTTP module
- Manage dependencies with npm

UNIT- IV: NODE JS

NodeJS Console

The Node.js Console refers to the command-line interface (CLI) or terminal environment where you can run Node.js commands and scripts.

Opening the Node.js Console

1. **Windows:**
    - Open **Command Prompt** or **PowerShell**.
    - Type node and press Enter.
2. **Mac/Linux:**
    - Open your terminal.
    - Type node and press Enter.

REPL Mode (Read-Eval-Print Loop)

Once you type node, you enter the **interactive Node.js console** (REPL mode). It allows you to run JavaScript line by line:

```
$ node
> 2 + 2
4
> console.log('Hello, Node.js!')
Hello, Node.js!
> .exit   # or press Ctrl+C twice to exit
```

Running Node.js Files

To run a Node.js script file:

1. Create a file named app.js:

```
console.log("Welcome to Node.js!");
```

2. Run it in the terminal:

```
node app.js
```

Output:

Welcome to Node.js!

Common Console Commands in Node.js with Examples

Command	Description	Example	Output
console.log()	Prints to stdout (standard output)	js console.log("Hello, Node.js!");	Hello, Node.js!
console.error()	Prints to stderr (error output)	js console.error("Something went wrong!");	Something went wrong! (in red)
console.warn()	Prints a warning message	js console.warn("This is a warning!");	This is a warning! (in yellow)
console.table()	Displays tabular data	js console.table([{ name: "Alice" }, { name: "Bob" }]);	Neatly formatted table
console.time() / timeEnd()	Measures execution time	js console.time("timer"); for(let i=0;i<100000;i++){} console.timeEnd("timer");	timer: X ms
console.clear()	Clears the console	js console.clear();	(Console gets cleared)

✅You can test all of these in a file like:

```
// app.js
console.log("Hello, Node.js!");
console.error("Something went wrong!");
console.warn("This is a warning!");
```

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

console.time("loop");
for (let i = 0; i < 1000000; i++) {}
console.timeEnd("loop");

console.clear(); // Comment this out to see earlier output
```

NodeJS Command Utilities

Basic Node.js Commands

Command	Description
node filename.js	Run a Node.js script file
node	Open Node.js REPL (interactive shell)
node --version or node -v	Show Node.js version installed
npm --version or npm -v	Show npm (Node Package Manager) version
npm init	Initialize a new package.json file interactively
npm init -y	Initialize package.json with default values
npm install package-name or npm i package-name	Install a package locally (in node_modules)
npm install -g package-name	Install a package globally

Command	Description
npm uninstall package-name	Remove a package
npm update	Update all packages
npm list or npm ls	List installed packages
npm outdated	Check for outdated packages
npm run script-name	Run custom scripts defined in package.json
npx command	Run package binaries without installing globally

Useful Node.js Utilities and Tools

1. nvm (Node Version Manager)

- Manage multiple Node.js versions on the same machine.
- Install nvm: https://github.com/nvm-sh/nvm
- Commands:
  - nvm install 18 (install Node.js v18)
  - nvm use 18 (switch to Node.js v18)
  - nvm ls (list installed Node versions)
  - nvm alias default 18 (set default Node version)

2. nodemon

- Utility that monitors changes in your source files and automatically restarts the Node.js app.
- Install: npm install -g nodemon
- Usage: nodemon app.js

3. pm2

- Process manager for Node.js apps, supports clustering, auto restarts, logs.

- Install: npm install -g pm2
- Basic usage:
    - pm2 start app.js
    - pm2 status
    - pm2 restart app
    - pm2 logs

4. eslint

- JavaScript linter to check code quality and style.
- Install: npm install -g eslint
- Initialize config: eslint --init
- Run linter: eslint yourfile.js

5. http-server

- Simple zero-configuration command-line HTTP server.
- Install: npm install -g http-server
- Usage: http-server ./public (serve static files)

Useful Node.js Debugging Commands

Command	Description
node --inspect filename.js	Start app with debugging enabled (Chrome DevTools can connect)
node --inspect-brk filename.js	Start app with debugging and break on first line
node debug filename.js	Start legacy debug mode (REPL debugging)
node --trace-warnings filename.js	Show stack traces on warnings

Miscellaneous Useful Commands

Command	Description
npm cache clean --force	Clear npm cache
npm doctor	Check environment and setup for common errors
npm audit	Run a security audit of your dependencies
npm audit fix	Fix vulnerabilities automatically
npm ci	Clean install node modules based on package-lock.json (for CI/CD)
npm start	Run the start script in package.json

NodeJS Modules

In Node.js, modules are reusable blocks of code whose scope is local by default but can be shared/exported and imported into other files. They help organize code into manageable pieces.

Types of Node.js Modules

1. Core Modules

Built-in modules shipped with Node.js. You don't need to install them separately.

Examples:

- fs — File system operations
- http — HTTP server/client
- path — File path utilities
- os — Operating system info
- events — Event emitter pattern
- crypto — Cryptography functions

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

2. Local Modules

Modules you create yourself within your project.

Example:
Create a file math.js:

```
function add(a, b) {
  return a + b;
}

module.exports = { add };
```

Use it in another file:

```
const math = require('./math');
console.log(math.add(2, 3)); // 5
```

3. Third-Party Modules

Modules installed via npm (Node Package Manager) from the npm registry.

Example: Installing and using express:

```
npm install express

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

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

app.listen(3000);
```

UNIT- IV: NODE JS

Module Systems in Node.js

CommonJS (used in Node.js by default)

1. Exporting a single function or object

```
// greet.js
module.exports = function() {
  return "Hello from CommonJS!";
};
```

Importing it:

```
// app.js
const greet = require('./greet');
console.log(greet());  // Output: Hello from CommonJS!
```

2. Exporting multiple things (properties)

```
// utils.js
module.exports = {
  sayHi: function() { return "Hi!"; },
  number: 123,
};
```

Importing it:

```
// app.js
const utils = require('./utils');
console.log(utils.sayHi());  // Output: Hi!
console.log(utils.number);   // Output: 123
```

3. Shortcut to export multiple properties

```
// helpers.js
exports.sayHello = function() { return "Hello!"; };
exports.value = 99;
```

ES Modules (modern JS using import/export)

1. Named exports (export multiple things by name)

```
// utils.js
export function sayHi() {
  return "Hi from ES Module!";
}
```

```
export const number = 456;
```

Importing named exports:

```
// app.js
import { sayHi, number } from './utils.js';
console.log(sayHi());  // Output: Hi from ES Module!
console.log(number);   // Output: 456
```

2. Default export (export a single default thing)

```
// greet.js
export default function() {
  return "Hello from default export!";
}
```

Importing default export:

```
// app.js
import greet from './greet.js';
console.log(greet());  // Output: Hello from default export!
```

Summary:

Concept	CommonJS	ES Modules
Export single	module.exports = fn	export default fn
Export multiple	module.exports = {} or exports.foo = ...	export function foo() {} or export const bar = ...
Import	const x = require()	import {x} from '...' or import x from '...'

NodeJS Concepts

1. What is Node.js?

- **Node.js** is a JavaScript runtime built on Chrome's V8 engine that lets you run JS code server-side.
- It uses an **event-driven, non-blocking I/O model** that makes it lightweight and efficient.

- Primarily used for building scalable network applications (web servers, APIs, real-time apps).

2. Asynchronous & Non-blocking I/O

- **Non-blocking I/O** means operations like reading files or making network requests don't block the main thread.
- Node.js uses **callbacks**, **Promises**, and **async/await** to handle these operations asynchronously.
- **Why important?** Prevents the application from freezing or waiting, enabling high concurrency.

Example (callback style):

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

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

- Output order:

  This prints before file content
  (then file content)

3. Event Loop

- The **event loop** is the core mechanism that allows Node.js to perform non-blocking I/O.
- Node.js runs on a **single thread**, but can handle multiple requests by putting tasks in an **event queue**.
- The event loop checks this queue and executes callbacks as soon as the call stack is empty.

Phases of Event Loop:

Phase	Description
Timers	Executes callbacks scheduled by setTimeout and setInterval
I/O callbacks	Executes some system I/O callbacks
Idle, prepare	Internal operations
Poll	Retrieves new I/O events; executes them
Check	Executes callbacks scheduled by setImmediate
Close callbacks	Executes close event callbacks (e.g., socket close)

- Example of difference between setImmediate and setTimeout:

```
setTimeout(() => console.log('timeout'), 0);
setImmediate(() => console.log('immediate'));
```

Output order can differ based on phase timing, but setImmediate often runs before zero-delay timeout.

4. Single Threaded but Highly Scalable

- Node.js handles multiple connections on a single thread using the event loop.
- Unlike traditional multi-threaded servers (like Apache), Node.js avoids the overhead of thread context switching.
- For CPU-heavy operations, Node.js offers **Worker Threads** or offloads tasks externally (e.g., to a database).

5. Modules

- Node.js uses the **CommonJS** module system.
- Each file is treated as a separate module with its own scope.
- Use require() to import and module.exports to export.

Example:

```
// greet.js
function sayHello(name) {
  return `Hello, ${name}!`;
}
module.exports = sayHello;

// app.js
const greet = require('./greet');
console.log(greet('Alice')); // Hello, Alice!
```

ES Modules (ESM) Support

- Node.js now also supports ES6 module syntax (import/export), but requires .mjs extension or "type": "module" in package.json.
- E.g.,

```
// greet.mjs
export function sayHello(name) {
  return `Hello, ${name}!`;
}

// app.mjs
import { sayHello } from './greet.mjs';
console.log(sayHello('Alice'));
```

6. Core Modules

Some core built-in modules to know:

Module	Description
fs	File system operations (read/write files)
http	Creating HTTP servers and clients
path	Handling file paths in a platform-independent way

Module	Description
os	Operating system info (CPU, memory, network)
crypto	Encryption, hashing, signing
events	EventEmitter implementation
stream	Handling streaming data (files, network, etc.)

7. NPM (Node Package Manager)

- Comes bundled with Node.js.
- Used for installing third-party libraries or publishing your own packages.
- Uses package.json to track dependencies, scripts, and metadata.
- Commands:
  - npm init → Create a new package.json.
  - npm install <package> → Installs and adds dependency.
  - npm update → Updates packages.
  - npm uninstall <package> → Removes dependency.

8. Creating HTTP Servers

Node.js allows building simple servers without frameworks:

```
const http = require('http');
const server = http.createServer((req, res) => {
  if (req.url === '/') {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.end('<h1>Welcome to Node.js server</h1>');
  } else {
    res.writeHead(404);
    res.end('Not found');
  }
});
```

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

9. Callbacks, Promises, Async/Await

Callbacks

- Basic async pattern but can cause “callback hell” if nested deeply.

Promises

- Represents a future value.
- Can be chained, improving readability.

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


fs.readFile('file.txt', 'utf8')
  .then(data => console.log(data))
  .catch(err => console.error(err));
```

Async/Await

- Syntactic sugar over promises, easier to read/write async code.

```
async function readFile() {
  try {
    const data = await fs.readFile('file.txt', 'utf8');
    console.log(data);
  } catch (err) {
    console.error(err);
  }
}
readFile();
```

10. EventEmitter

- Node.js uses **EventEmitter** to handle events.
- Many core modules (like http) inherit from EventEmitter.

Example:

```
const EventEmitter = require('events');
const emitter = new EventEmitter();

emitter.on('start', () => {
  console.log('Started');
});

emitter.emit('start');
```

11. Streams

- Streams allow processing data piece-by-piece instead of all at once.
- Useful for large files or continuous data (videos, audio, network packets).

Types:

- **Readable:** source you can read from (fs.createReadStream)
- **Writable:** destination you can write to (fs.createWriteStream)
- **Duplex:** both readable and writable (network sockets)
- **Transform:** modifies data as it passes through

Example of reading a file stream:

```
const fs = require('fs');
const readStream = fs.createReadStream('largefile.txt', 'utf8');

readStream.on('data', chunk => {
  console.log('Received chunk:', chunk);
});
```

```
readStream.on('end', () => {
  console.log('Finished reading');
});
```

12. Buffer

- Buffers are used to work with raw binary data.
- Can be created from strings or arrays.

```
const buf = Buffer.from('Hello');
console.log(buf); // <Buffer 48 65 6c 6c 6f>
```

Useful for handling network protocols, file I/O, encryption.

13. Process & Global Objects

- process provides info and control over the running Node.js process:
  - process.argv — Command-line arguments.
  - process.env — Environment variables.
  - process.exit() — Exit the process.
  - Events: exit, uncaughtException, SIGINT.

Example:

```
console.log(process.argv);
```

- Global objects like __dirname, __filename, global, console.

14. Error Handling

- Synchronous code: use try/catch.
- Async callbacks: handle errors in callback parameters ((err, result) => {}).
- Promises: use .catch() or try/catch with async/await.
- Handle uncaught exceptions globally:

```
process.on('uncaughtException', (err) => {
  console.error('Unhandled Exception', err);
  process.exit(1); // recommended to restart
```

```
});
```

15. Worker Threads (Advanced)

- For CPU-intensive tasks that block the event loop.
- Allows multi-threading in Node.js.

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

const worker = new Worker('./worker.js');

worker.on('message', message => console.log(message));
worker.postMessage('start');
```

16. Debugging & Profiling

- Use built-in debugger:

  node inspect app.js

- Or use Chrome DevTools:

  node --inspect app.js

- Profiling tools: clinic, node --prof.

NodeJS Events

1. What Are Events in Node.js?

- Node.js uses an **event-driven architecture**.
- An **event** is a signal that something happened — like a user click, a server request, or a file read completion.
- Node.js apps listen for events and react when they occur.
- This is core to Node.js's **non-blocking, asynchronous** behavior.

2. EventEmitter Class

- The main way to work with events in Node.js is through the **EventEmitter** class from the events module.
- Almost all Node.js core modules (like HTTP, Streams, Sockets) inherit from EventEmitter.
- An **EventEmitter** object emits named events that cause functions ("listeners" or "handlers") to be called.

3. Basic Usage of EventEmitter

Import the module and create an emitter:

```
const EventEmitter = require('events');
const emitter = new EventEmitter();
```

Register an event listener (subscribe):

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

Emit the event (publish):

```
emitter.emit('greet', 'Alice');  // Output: Hello, Alice!
```

- on() listens for every time the event occurs.
- emit() triggers the event and calls all listeners registered for it.

4. Common Methods of EventEmitter

Method	Description
on(event, fn)	Register a listener for the event
once(event, fn)	Register a one-time listener; auto-removed after first call
emit(event, [...args])	Emit the event, calling all listeners with optional args

Method	Description
removeListener(event, fn)	Remove a specific listener for an event
removeAllListeners(event)	Remove all listeners for an event
listenerCount(event)	Get the number of listeners registered for an event
listeners(event)	Returns an array of listeners for an event

5. once() vs on()

- on() will call the listener every time the event fires.
- once() will call the listener only the **first time** the event fires, then remove it.

Example:

```
emitter.once('init', () => console.log('Initialization done'));
emitter.emit('init'); // Prints once
emitter.emit('init'); // Does nothing
```

6. Handling Errors

- error is a special event.
- If an error event is emitted and there is no listener for it, Node.js will **throw and crash**.
- Always add an error listener to catch errors gracefully.

Example:

```
emitter.on('error', (err) => {
  console.error('Error occurred:', err.message);
});


emitter.emit('error', new Error('Something went wrong'));
```

7. Example: Custom Event Emitter

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

```
class MyEmitter extends EventEmitter {}

const myEmitter = new MyEmitter();

// Listener function
myEmitter.on('start', () => {
  console.log('Started processing');
});

myEmitter.on('data', (data) => {
  console.log('Data received:', data);
});

myEmitter.on('end', () => {
  console.log('Processing ended');
});

// Emit events
myEmitter.emit('start');
myEmitter.emit('data', 'Some payload');
myEmitter.emit('end');
```

8. EventEmitter Max Listeners

- By default, an EventEmitter warns if more than **10 listeners** are added to the same event to avoid potential memory leaks.
- You can change this limit with:

```
emitter.setMaxListeners(20);
```

- Or disable the warning:

```
emitter.setMaxListeners(0);
```

9. Event Propagation and prependListener

- You can register a listener that runs **before** others with:

```
emitter.prependListener('event', listenerFn);
```

- Similar to on(), but called before existing listeners.

10. Real-World Uses of Events in Node.js

- **HTTP server:** Emits events like 'request', 'close'.
- **Streams:** Emit 'data', 'end', 'error'.
- **File system:** Emit 'open', 'close', 'change'.
- **Child processes:** Emit 'exit', 'error'.
- **Custom modules:** Implement event emitters to notify about async operations.

11. Event Loop & EventEmitter

- The **event loop** manages when event callbacks run.
- When an event is emitted, its listeners are queued and run on the next tick of the event loop.
- EventEmitter is the basis for Node's asynchronous event-driven I/O model.

12. Advanced: Using once with Promises (Node.js v11+)

Node.js provides a built-in way to wait for an event once, using events.once:

```
const { once } = require('events');
async function run() {
  const myEmitter = new EventEmitter();

  setTimeout(() => myEmitter.emit('finish'), 1000);

  await once(myEmitter, 'finish');
  console.log('Finished event detected');
}
run();
```

UNIT- IV: NODE JS

Summary

- Events are the heart of Node.js's asynchronous programming.
- EventEmitter class lets you create, emit, and listen for events.
- Events enable modular, decoupled, and scalable code.
- Always handle 'error' events to prevent app crashes.
- Use once() to listen for single-time events.
- Understand the event loop to grasp how and when event callbacks execute.

NodeJS with ExpressJS

What is Express.js?

- **Express.js** is a minimal and flexible **web application framework** for Node.js.
- It simplifies building APIs and web apps by handling:
  - Routing
  - Middleware
  - Request & Response handling
  - Templating (if needed)
- It's **built on top of Node.js's HTTP module**, but with much less boilerplate.

Installing Express

```
npm init -y
npm install express
```

Basic Express Server

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

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

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

```
});
```

Core Concepts of Express.js

1. Routing

- Define how your app responds to HTTP requests at different endpoints and methods.

```
app.get('/users', (req, res) => res.send('Get users'));
app.post('/users', (req, res) => res.send('Create user'));
app.put('/users/:id', (req, res) => res.send(`Update user ${req.params.id}`));
app.delete('/users/:id', (req, res) => res.send(`Delete user ${req.params.id}`));
```

2. Middleware

- Functions that execute **in sequence** during the request lifecycle.
- Can:
  - Modify req, res
  - End the request-response cycle
  - Call the next middleware

Types of Middleware:

- Application-level
- Router-level
- Error-handling
- Built-in (express.json(), express.static())
- Third-party (e.g. cors, morgan)

```
app.use(express.json()); // Built-in middleware to parse JSON body
app.use((req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next();
});
```

3. Request and Response

- req (request): Contains HTTP request info (headers, query params, body, etc.)
- res (response): Used to send a response (HTML, JSON, status codes)

```
app.post('/login', (req, res) => {
  const { username, password } = req.body;
  res.status(200).json({ message: `Welcome, ${username}` });
});
```

4. Serving Static Files

```
app.use(express.static('public')); // Now you can access public/index.html
```

5. Routing with Parameters

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

6. Route Grouping using Router

```
const userRouter = express.Router();

userRouter.get('/', (req, res) => res.send('All users'));
userRouter.post('/', (req, res) => res.send('Create user'));

app.use('/api/users', userRouter);
```

Connecting with MongoDB (via Mongoose)

```
npm install mongoose
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost:27017/testdb')
  .then(() => console.log('MongoDB connected'))
  .catch(err => console.error('MongoDB error', err));
```

Handling Errors

```
app.use((err, req, res, next) => {
  console.error(err.stack);
```

```
  res.status(500).send('Something broke!');
});
```

Project Structure Example

```
project/
├── app.js
├── routes/
│   └── userRoutes.js
├── controllers/
│   └── userController.js
├── models/
│   └── User.js
├── middleware/
│   └── authMiddleware.js
```

Sample Features with Express

✅Authentication (Login example):

```
app.post('/login', (req, res) => {
  const { username, password } = req.body;
  // Dummy check
  if (username === 'admin' && password === '123') {
    return res.json({ token: 'dummy-jwt-token' });
  }
  res.status(401).send('Unauthorized');
});
```

Sending JSON Response

```
app.get('/json', (req, res) => {
  res.json({ name: "Express", type: "Framework" });
});
```

⏳ Middlewares for Logging

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

```
app.use(morgan('dev'));  // logs requests like: GET /users 200 10ms
```

Handling 404

```
app.use((req, res) => {
  res.status(404).send('Page Not Found');
});
```

Summary Table

Concept	Express.js Equivalent
HTTP Server	app.listen(port, callback)
Routing	app.get(), app.post(), etc.
Middleware	app.use()
Static Files	express.static()
Parsing JSON	express.json()
Modular Routing	express.Router()
Error Handling	Custom middleware with 4 params
MongoDB Support	Via mongoose

NodeJS Database Access

Node.js can interact with **any type of database** using appropriate drivers or ORMs.

Types of Databases

Type	Examples	Data Model	Use Case
SQL	MySQL, PostgreSQL, SQLite	Table/Rows	Structured, relational data

Type	Examples	Data Model	Use Case
NoSQL	MongoDB, Redis, Cassandra	Document/Key-Value	Flexible, unstructured data

1. MongoDB (NoSQL)

Setup

npm install mongoose

Connecting with Mongoose

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

mongoose.connect('mongodb://127.0.0.1:27017/testdb', {
  useNewUrlParser: true,
  useUnifiedTopology: true
})
.then(() => console.log('MongoDB Connected'))
.catch(err => console.error('Mongo Error:', err));
```

Creating a Schema and Model

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

const User = mongoose.model('User', userSchema);
```

CRUD Operations

```
// Create
await User.create({ name: 'John', email: 'john@example.com', age: 25 });
```

```
// Read
const users = await User.find({ age: { $gt: 18 } });

// Update
await User.updateOne({ name: 'John' }, { age: 26 });

// Delete
await User.deleteOne({ name: 'John' });
```

2. MySQL (SQL)

Setup

```
npm install mysql2
```

Connect to MySQL

```
const mysql = require('mysql2');

const connection = mysql.createConnection({
  host: 'localhost',
  user: 'root',
  password: '',
  database: 'testdb'
});

connection.connect(err => {
  if (err) throw err;
  console.log('MySQL Connected');
});
```

CRUD Operations

```
// Create
connection.query(
```

```
  'INSERT INTO users (name, email) VALUES (?, ?)',
  ['Alice', 'alice@example.com'],
  (err, results) => {
    if (err) throw err;
    console.log('Inserted:', results.insertId);
  }
);

// Read
connection.query('SELECT * FROM users', (err, rows) => {
  console.log(rows);
});

// Update
connection.query(
  'UPDATE users SET name = ? WHERE id = ?',
  ['Bob', 1],
  (err, result) => {
    console.log('Updated:', result.affectedRows);
  }
);

// Delete
connection.query('DELETE FROM users WHERE id = ?', [1]);
```

3. PostgreSQL

Setup

```
npm install pg
```

Connect and Query

```
const { Client } = require('pg');
```

```
const client = new Client({
  user: 'postgres',
  host: 'localhost',
  database: 'testdb',
  password: 'password',
  port: 5432
});

client.connect()
  .then(() => console.log('PostgreSQL Connected'))
  .catch(err => console.error(err));

// Query
client.query('SELECT * FROM users', (err, res) => {
  console.log(res.rows);
});
```