

1. Basics of JavaScript (Pre-requisite)

Before diving into Node.js, you must be comfortable with:

- ES6+ syntax (let, const, arrow functions, destructuring)
- Promises, `async/await`
- Array methods (map, filter, reduce)
- Functions, callbacks, closures, `this`
- Modules (`import/export`)

2. Introduction to Node.js

- What is Node.js?
- Node.js vs traditional backend (like PHP)
- V8 Engine and how Node runs JS outside browser
- Event loop and non-blocking I/O
- Use cases: APIs, microservices, real-time apps

Activity: Install Node.js, run `console.log("Hello Node")`

3. Core Modules in Node.js

- `fs` – File system (read/write files)
- `path` – Work with file paths
- `os` – Get OS info
- `http` – Build simple servers

Activity: Create a basic HTTP server using `http` module.

4. NPM and Modules

- What is NPM (Node Package Manager)

- Local vs global packages
- Installing third-party packages (`express`, `nodemon`)
- Semantic Versioning
- Creating your own modules

Activity: Build a small CLI app using npm package like `inquirer`

5. Express.js (Node.js Framework)

- What is Express?
- Creating API routes (GET, POST, PUT, DELETE)
- Route parameters & query params
- Middleware & error handling
- Static file serving
- RESTful API design

Activity: Build a CRUD API for a Todo app.

6. Working with Databases

- **MongoDB + Mongoose**
 - Connect to MongoDB using Mongoose
 - Schemas and Models
 - CRUD operations
- (Optionally) **MySQL/PostgreSQL with Sequelize/Knex.js**

Activity: Connect your Todo API to MongoDB.

7. Authentication and Authorization

- User registration and login
- Password hashing using `bcrypt`
- JSON Web Tokens (JWT) for auth

- Protecting routes (middleware)
- Role-based access

Activity: Add login/register & JWT auth to your API.

8. File Uploads

- Uploading files using multer

1. ES6+ Syntax

let and const

Used to declare variables.

```
let age = 25;    // can be reassigned
const pi = 3.14; // cannot be reassigned
```

Difference from var:

- var is function-scoped.
- let and const are block-scoped.

```
{
  let x = 10;
  var y = 20;
}
console.log(y); // ☑ 20
```

```
console.log(x); // X Error: x is not defined
```

Arrow Functions

Shorter syntax for writing functions.

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

// Arrow function
const add = (a, b) => a + b;

console.log(add(2, 3)); // 5
```

Arrow functions do **not** have their own `this`.

Destructuring

Extract values from arrays or objects into variables.

```
// Array destructuring
const nums = [10, 20];
const [a, b, c] = nums;
console.log(a, b); // 10 20

// Object destructuring
const person = { name: "Alice", age: 30 };
const { name, age } = person;
console.log(name, age); // Alice 30
```

2. Promises and async/await

Promises

A way to handle asynchronous operations like fetching data.

It represents object

States in promise

```
1.pending  
2.fullfilled  
3.rejected
```

Example

```
const getData = () => {  
  return new Promise((resolve, reject) => {  
    setTimeout(() => {  
      resolve("Data received");  
    }, 1000);  
  });  
};  
  
getData().then(data => console.log(data)); // Data received  
  
// !promises  
  
console.log("this is second")  
  
  
new Promise((resolve,reject)=>{  
  let data=''  
  if (data.length>0){  
    resolve(data)  
  }  
})
```

```
        else{
            reject('data no fetched')
        }

    })
    .then((data1)=>{
        console.log(data1)
    }).catch((error)=>{
        console.log(error)
    })

// !json

let obj={
    id:1,
    name:'mohan'
}
let obj1=JSON.stringify(obj)
console.log(obj1)

console.log(JSON.parse(obj1))

// !async

console.log("this is first")

const getData= async ()=>{
    fetch('https://fakestoreapi.com/products/1').then((res)=>{
        return res.json()
    }).then((data)=>console.log(data))
    .catch((error)=>{
        console.log(error)
    })
}

getData()
console.log('this is second')
```

async/await

Simplifies working with Promises.

Example1:

```
//!async and await

let getData=async ()=>{
  let res=await fetch('https://fakestoreapi.com/products/1')
  let data= await res.json()
  console.log(data)
}

getData()

Console.log()
```

Example2:

```
const fetchData = async () => {
  const data = await getData(); // wait for promise to resolve
  console.log(data);
};

fetchData(); // Data received
```

3. Array Methods

map()

Transforms each element and returns a **new array**.

```
const nums = [1, 2, 3];
const doubled = nums.map(n => n * 2);
console.log(doubled); // [2, 4, 6]
```

filter()

Filters out elements based on a condition.

```
const nums = [1, 2, 3, 4];
const evens = nums.filter(n => n % 2 === 0);
console.log(evens); // [2, 4]
```

reduce()

Reduces array to a single value (sum, average, etc.)

```
const nums = [1, 2, 3, 4];
const sum = nums.reduce((acc, curr) => acc + curr, 0);
console.log(sum); // 10
```

4. Functions, Callbacks, Closures, this

Functions & Callbacks

Callback: A function passed as an argument to another function.

```
function greet(name) {  
  console.log("Hello", name);  
}  
  
function processUser(callback) {  
  const name = "Mohan";  
  callback(name);  
}  
  
processUser(greet); // Hello Mohan
```

Closures

Function **remembers** variables from its parent scope.

```
function outer() {  
  let count = 0;  
  
  function inner() {  
    count++;  
    console.log(count);  
  }  
  
  return inner;  
}  
  
const counter = outer();  
counter(); // 1  
counter(); // 2
```

this Keyword

Refers to the object that is **calling** the function.

```
const user = {  
  name: "Mohan",  
  greet() {  
    console.log("Hello, " + this.name);  
  }  
};  
  
user.greet(); // Hello, Mohan
```

Arrow functions **don't bind this**, they use the **this** of their surrounding context.

5. Modules: import/export

Used to organize code across multiple files.

Exporting from a file (math.js)

```
export default const add = (a, b) => a + b;  
export const subtract = (a, b) => a - b;
```

Importing into another file

```
import sum,{ subtract } from './math.js';  
  
console.log(add(5, 3)); // 8
```

If using Node.js with **require**, use CommonJS:

```
// math.js
module.exports = { add, subtract };

// main.js
const { add } = require('./math');
```

1. Introduction to Node.js (With Detailed Explanation + Examples)

What is Node.js?

Node.js is an **open-source, cross-platform runtime environment** that allows you to run **JavaScript code on the server**.

Breakdown:

- **JavaScript** was originally made to run in the browser only.
- **Node.js** allows developers to use JavaScript to build **backend (server-side)** logic like:
 - Handling user requests
 - Reading/writing files
 - Connecting to databases
 - Sending emails
 - Running background tasks

Example:

```
// app.js  
console.log("Server is starting...");
```

When you run this with Node:

```
node app.js
```

Output:

```
Server is starting...
```

Node runs JavaScript **outside the browser**, directly in your terminal.

Why use Node.js?

Here are the **main benefits** of using Node.js:

1. Same Language for Frontend and Backend

You use **JavaScript on both sides** — frontend (browser) and backend (server). This simplifies development.

Before:

- Frontend: JavaScript
- Backend: PHP, Java, Python

Now (with Node.js):

- Frontend: JavaScript
- Backend: JavaScript (Node.js)

This means: one language, less context-switching, faster learning.

2. ⚡ Asynchronous and Non-blocking

Node.js uses a **non-blocking** model, which means it **does not wait** for operations like file reading or API calls to finish. Instead, it moves on and handles the response **later**.

- This makes Node.js very **fast** for apps that handle many users or requests.

Example:

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

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

console.log("Reading file...");
```

Output:

```
Reading file...
(Then shows file content later)
```

3. Fast Execution with V8 Engine

Node.js uses Google Chrome's **V8 JavaScript engine**, which is highly optimized and compiles JavaScript to machine code — resulting in faster performance.

4. Huge Ecosystem via npm

npm (Node Package Manager) offers over **1 million packages** — ready-to-use libraries for almost anything.

Example:

- express: web framework
- mongoose: MongoDB library
- dotenv: environment variable management
- bcrypt: password hashing

5. Real-time Applications

Node.js is ideal for **real-time apps** like:

- Chat applications (WhatsApp, Messenger)
- Online games (multiplayer)
- Live stock market feeds
- Collaborative tools (Google Docs-like)

Node.js vs Traditional Backend Tech (PHP, Java)

Feature	Node.js	PHP / Java
Language	JavaScript	PHP / Java
Execution Style	Asynchronous (non-blocking)	Synchronous (blocking)
Performance	Very fast in I/O-heavy apps	Slightly slower in real-time apps
Development Speed	Fast with npm & JS knowledge	Slower (more setup & learning)
Real-time App Support	Excellent with WebSockets	Not built-in, needs extra config
Learning Curve	Easier if frontend dev knows JS	Requires learning new language

V8 Engine and Event Loop (Basic Explanation)

V8 Engine

- Created by Google for Chrome browser.
- It converts JavaScript code into **machine code**, which the system understands.
- Node.js uses this same engine for **fast server-side JavaScript execution**.

Think of V8 as the “engine” in a race car — it makes the JavaScript run blazing fast.

Event Loop

Node.js runs on **a single thread**, but it can handle **thousands of requests at the same time** using the **event loop**.

Here's how:

1. Node starts a task (e.g., reading a file)
2. While waiting, it keeps doing other tasks
3. When the file is ready, it returns the result

This is called **non-blocking I/O**, and it's what makes Node.js fast and scalable.

Simple Code Example:

```
setTimeout(() => {
  console.log("Timeout finished!");
}, 2000);

console.log("First line");
```

Output:

```
First line
(2 seconds later)
Timeout finished!
```

Even though `setTimeout` waits 2 seconds, Node continues running the rest of the code.

Use Cases of Node.js

Here's where Node.js really shines:

1. Chat Applications

Real-time chatting with WebSockets (e.g., WhatsApp clone)

2. RESTful APIs

Build backend APIs for mobile or web apps (e.g., login, get data, post comments)

3. Data Streaming

Netflix, YouTube use Node.js to stream large video/audio files

4. Microservices

Break large apps into smaller parts (used in Uber, Netflix, etc.)

5. IoT and Device Control

Connect devices using Node.js for home automation, sensors, etc.

Activity: Install Node.js and Run Your First Program

Step 1: Install Node.js

- Visit: <https://nodejs.org>
- Download and install the **LTS version**

Step 2: Check Installation

Open terminal / command prompt:

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

You'll see versions like v20.11.1 and 10.x.x

Step 3: Create and Run a File

Create a file named app.js and add:

```
console.log("Hello Node");
```

Run it:

```
node app.js
```

Output:

```
Hello Node
```

Congratulations! 🎉 You've just written and executed your first Node.js code.

fs – File System Module

What is it?

The fs module in Node.js allows you to **interact with the file system** — that means you can **read, write, append, delete, rename, copy files**, and also **work with directories**.

It supports both:

- **Asynchronous (non-blocking)** functions — recommended for most use-cases
- **Synchronous (blocking)** functions — useful for small scripts or setup

To use it:

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

Common Operations in fs

1. Write to a File

Asynchronous:

```
fs.writeFile('hello.txt', 'Hello World!', (err) => {
  if (err) throw err;
  console.log('File created and written successfully.');
});
```

Synchronous:

```
fs.writeFileSync('hello.txt', 'Hello World!');
```

If the file exists, it will overwrite. If not, it creates it.

2. Append to a File

Asynchronous:

```
fs.appendFile('hello.txt', '\nAppended line.', (err) => {
  if (err) throw err;
  console.log('Data appended to file.');
});
```

Synchronous:

```
fs.appendFileSync('hello.txt', '\nAppended line.');
```

3. Read from a File

Asynchronous:

```
fs.readFile('hello.txt', 'utf-8', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

Synchronous:

```
const content = fs.readFileSync('hello.txt', 'utf-8');
console.log(content);
```

4. Rename a File

```
fs.rename('hello.txt', 'greetings.txt', (err) => {
  if (err) throw err;
```

```
    console.log('File renamed.');
});
```

5. Delete a File

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

6. Check if File Exists

```
if (fs.existsSync('hello.txt')) {
  console.log('File exists');
} else {
  console.log('File does not exist');
}
```

7. Create a Directory

Asynchronous:

```
fs.mkdir('myFolder', (err) => {
  if (err) throw err;
  console.log('Directory created.');
});
```

With recursive option:

```
fs.mkdir('parent/child/grandchild', { recursive: true }, (err) => {
  if (err) throw err;
  console.log('Nested folders created.');
```

```
});
```

8. Read Directory Contents

```
fs.readdir('.', (err, files) => {
  if (err) throw err;
  console.log('Files in current directory:', files);
});
```

9. Delete a Directory

```
fs.rmdir('myFolder', (err) => {
  if (err) throw err;
  console.log('Directory deleted.');
});
```

②`fs.rmdir()` is deprecated. Use `fs.rm()` with `recursive: true`.

```
fs.rm('myFolder', { recursive: true, force: true }, (err) => {
  if (err) throw err;
  console.log('Directory removed.');
});
```

10. Copy a File

```
fs.copyFile('source.txt', 'destination.txt', (err) => {
  if (err) throw err;
  console.log('File copied.');
});
```

11. File Stats (Size, Created Time, etc.)

```
fs.stat('hello.txt', (err, stats) => {
  if (err) throw err;
  console.log('Is file:', stats.isFile());
  console.log('Size:', stats.size);
  console.log('Created:', stats.birthtime);
});
```

os – Operating System Module in Node.js

What is os?

The os module in Node.js provides **operating system-related utility methods and properties**.

It helps you gather **info about the host system** (CPU, memory, platform, architecture, etc.).

It's a **core module**, so you don't need to install anything extra.

Syntax:

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

Key Functions and Examples

1. os.arch()

Returns the **CPU architecture** of the system.

```
console.log(os.arch());
```

Possible outputs:

x64

arm

2. os.platform()

Returns the **operating system platform**.

```
console.log(os.platform());
```

Possible outputs:

```
win32      // Windows
linux      // Linux
darwin     // macOS
```

3. os.type()

Returns the **OS name** (like 'Linux', 'Windows_NT').

```
console.log(os.type());
```

Example output:

```
Linux
Windows_NT
```

4. os.hostname()

Returns the **host name** of the system.

```
console.log(os.hostname());
```

Output example:

Mohan-PC

5. os.totalmem()

Returns the **total system memory** in **bytes**.

```
console.log(os.totalmem());
```

Output:

8372142080 // (Bytes) = ~8 GB

You can convert to GB:

```
console.log((os.totalmem() / 1024 / 1024 / 1024).toFixed(2) + ' GB');
```

6. os.freemem()

Returns the **free memory** available (also in bytes).

```
console.log(os.freemem());
```

Convert to MB or GB for readability:

```
console.log((os.freemem() / 1024 / 1024).toFixed(2) + ' MB');
```

7. os.uptime()

Returns **system uptime** in **seconds**.

```
console.log(os.uptime());
```

Convert to hours/mins:

```
const uptime = os.uptime();
console.log(`Uptime: ${Math.floor(uptime / 3600)} hours`);
```

8. os.cpus()

Returns an array of **CPU core information**.

```
console.log(os.cpus());
```

Each object in the array contains:

- Model
- Speed (MHz)
- Times (user, system, idle, etc.)

Example output:

```
[  
  {  
    model: 'Intel(R) Core(TM) i5...',  
    speed: 2400,  
    times: {  
      user: 100000,  
      system: 20000,  
      idle: 500000,  
      ...  
    }  
  }]
```

```
},  
...  
]
```

To see the number of cores:

```
console.log('CPU cores:', os.cpus().length);
```

9. os.networkInterfaces()

Returns the **network interfaces** (like eth0, Wi-Fi, lo).

```
console.log(os.networkInterfaces());
```

It shows:

- IPv4 and IPv6 addresses
- MAC address
- Internal (localhost) or external

10. os.userInfo()

Returns info about the **current logged-in user**.

```
console.log(os.userInfo());
```

Output:

```
{  
  uid: -1,  
  gid: -1,  
  username: 'mohan',  
  homedir: '/Users/mohan',  
  shell: '/bin/bash'
```

```
}
```

11. os.tmpdir()

Returns the path to the **temporary directory** used by the OS.

```
console.log(os.tmpdir());
```

Example:

```
C:\Users\Mohan\AppData\Local\Temp
```

12. os.endianness()

Returns CPU **endianness** (BE for Big Endian, LE for Little Endian).

```
console.log(os.endianness()); // 'LE'
```

Full Example Program

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

console.log('Platform      :', os.platform());
console.log('OS Type       :', os.type());
console.log('Architecture  :', os.arch());
console.log('CPU Cores     :', os.cpus().length);
console.log('CPU Model     :', os.cpus()[0].model);
console.log('Total Memory   :', (os.totalmem() / 1024 / 1024 /
1024).toFixed(2), 'GB');
console.log('Free Memory    :', (os.freemem() / 1024 /
1024).toFixed(2), 'MB');
console.log('Uptime         :', (os.uptime() / 3600).toFixed(2),
```

```

'hours');

console.log('Home Dir      :', os.homedir());
console.log('Temp Dir       :', os.tmpdir());
console.log('Host Name      :', os.hostname());
console.log('User Info       :', os.userInfo().username);
console.log('Network Info   :', JSON.stringify(os.networkInterfaces(),
null, 2));

```

Summary Table

Method	Description
os.arch()	CPU architecture (e.g., x64)
os.platform()	Platform (win32, linux, darwin)
os.type()	OS name (Windows_NT, Linux)
os.totalmem()	Total system memory (bytes)
os.freemem()	Free memory available
os.uptime()	Uptime (in seconds)
os.hostname()	Hostname of the system
os.cpus()	CPU core details
os.networkInterfaces()	Network interface information
os.userInfo()	Current user info
os.tmpdir()	OS temp directory path
os.endianness()	CPU byte order

path – Node.js Path Module

What is path?

The path module in Node.js provides **utilities for working with file and directory paths**. It ensures that your path-related operations are **cross-platform compatible** — important because:

- Windows uses \ (backslash)
- macOS/Linux use / (forward slash)

It's a **core module** — no need to install anything.

How to import:

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

Important Methods with Definitions, Syntax, Examples, and Output

1. **path.join([...paths])**

Definition: Joins multiple path segments into one normalized path.

Syntax:

```
path.join(path1, path2, ..., pathN)
```

Example:

```
console.log(path.join('users', 'mohan', 'notes.txt'));
```

Output (Windows):

```
users\mohan\notes.txt
```

Output (Linux/macOS):

```
users/mohan/notes.txt
```

Automatically handles slashes.

2. path.resolve([...paths])

Definition: Resolves a sequence of paths into an **absolute path**.

Example:

```
console.log(path.resolve('folder', 'file.txt'));
```

Output:

```
C:\Users\You\folder\file.txt (Windows)
```

Starts from the current working directory and resolves paths from right to left.

3. path.basename(path, [ext])

Definition: Returns the **last part of a path** (usually the filename).

Example:

```
console.log(path.basename('/users/mohan/docs/file.txt'));           //  
file.txt  
console.log(path.basename('/users/mohan/docs/file.txt', '.txt')); //  
file
```

4. path.dirname(path)

Definition: Returns the **directory name** of a path (everything except the file).

Example:

```
console.log(path.dirname('/users/mohan/docs/file.txt'));
```

Output:

```
/users/mohan/docs
```

5. path.basename(path)

Definition: Returns the **file extension**, including the dot.

Example:

```
console.log(path.basename('notes.md')); // .md
```

6. path.parse(path)

Definition: Breaks a path into its components.

Example:

```
console.log(path.parse('/users/mohan/docs/file.txt'));
```

Output:

```
{
  root: '/',
  dir: '/users/mohan/docs',
  base: 'file.txt',
  ext: '.txt',
```

```
    name: 'file'  
}  
  
}
```

7. path.format(pathObject)

Definition: Builds a path from an object — **opposite of path.parse()**.

Example:

```
console.log(path.format({  
  dir: '/users/mohan/docs',  
  name: 'file',  
  ext: '.txt'  
}));
```

Output:

```
/users/mohan/docs/file.txt
```

8. path.isAbsolute(path)

Definition: Returns true if the path is absolute, otherwise false.

Example:

```
console.log(path.isAbsolute('/file.txt'));      // true  
console.log(path.isAbsolute('folder/file.txt')); // false
```

9. path.relative(from, to)

Definition: Returns the **relative path** from one location to another.

Example:

```
console.log(path.relative('/users/mohan',
  '/users/mohan/docs/file.txt'));
```

Output:

docs/file.txt

10. path.sep

Definition: Returns the **platform-specific separator** (/ or \).

```
console.log(path.sep); // '\' on Windows, '/' on POSIX
```

11. path.delimiter

Definition: Returns the **delimiter used in PATH environment variable**.

```
js
CopyEdit
console.log(path.delimiter); // ';' on Windows, ':' on POSIX
```

Full Example Program

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

const filePath = '/users/mohan/docs/file.txt';

console.log('Path      :', filePath);
console.log('Base Name  :', path.basename(filePath));
console.log('Dir Name   :', path.dirname(filePath));
console.log('Extension  :', path.extname(filePath));
console.log('Is Absolute?:', path.isAbsolute(filePath));
```

```

console.log('Parsed      :', path.parse(filePath));
console.log('Formatted    :', path.format(path.parse(filePath)));
console.log('Join Path    :', path.join('folder', 'images',
'img.jpg'));
console.log('Resolve Path :', path.resolve('folder', 'images',
'img.jpg'));
console.log('Relative Path:', path.relative('/users/mohan',
filePath));
console.log('Path Sep     :', path.sep);
console.log('Path Delim   :', path.delimiter);

```

Summary Table

Method	Description
path.join()	Join multiple path segments
path.resolve()	Resolve to an absolute path
path.basename()	Get the file name from a path
path.dirname()	Get the directory part of a path
path.extname()	Get file extension
path.parse()	Break path into parts
path.format()	Assemble a path from an object
path.isAbsolute()	Check if a path is absolute
path.relative()	Get relative path between two locations
path.sep	Platform-specific separator (/ or \)
path.delimiter	Separator used in PATH variable (: or ;)

http Module in Node.js

What is the http Module?

The `http` module in Node.js allows you to:

- Create a basic web server
- Handle HTTP requests and responses (like GET, POST, PUT, DELETE)
- Listen on a specific port (like 3000 or 8080)

How to use it:

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

const server = http.createServer((req, res) => {
  // request: incoming HTTP request (method, headers, URL, etc.)
  // response: what to send back to client
});

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

Common HTTP Methods

Method	Description
GET	Retrieve data (e.g., fetch a user)
POST	Send new data (e.g., create a user)
PUT	Update existing data
DELETE	Remove data

Full Example: Handling GET, POST, PUT, DELETE

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

```
let users = [
  { id: 1, name: "Mohan" },
  { id: 2, name: "Niranjan" }
];

const server = http.createServer((req, res) => {
  const parsedUrl = url.parse(req.url, true);
  const { pathname, query } = parsedUrl;

  if (pathname === "/users" && req.method === "GET") {
    // GET all users
    res.writeHead(200, { "Content-Type": "application/json" });
    res.end(JSON.stringify(users));
  }

  else if (pathname === "/users" && req.method === "POST") {
    // POST create user
    let body = "";
    req.on("data", chunk => body += chunk);
    req.on("end", () => {
      const newUser = JSON.parse(body);
      newUser.id = users.length + 1;
      users.push(newUser);

      res.writeHead(201, { "Content-Type": "application/json" });
      res.end(JSON.stringify(newUser));
    });
  }

  else if (pathname.startsWith("/users/") && req.method === "PUT") {
    // PUT update user
    const id = parseInt(pathname.split("/")[2]);
    let body = "";
    req.on("data", chunk => body += chunk);
    req.on("end", () => {
      const updatedUser = JSON.parse(body);
      users = users.map(u => u.id === id ? { ...u, ...updatedUser } : u);
    });
  }
}
```

```

        res.writeHead(200, { "Content-Type": "application/json" });
        res.end(JSON.stringify({ message: "User updated" })));
    });
}

else if (pathname.startsWith("/users/") && req.method === "DELETE")
{
    // DELETE a user
    const id = parseInt(pathname.split("/")[2]);
    users = users.filter(u => u.id !== id);

    res.writeHead(200, { "Content-Type": "application/json" });
    res.end(JSON.stringify({ message: "User deleted" }));
}

else {
    res.writeHead(404, { "Content-Type": "application/json" });
    res.end(JSON.stringify({ message: "Route not found" }));
}
});

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

```

Test it using Postman or curl:

GET all users:

```
curl http://localhost:3000/users
```

POST (create user):

```
curl -X POST http://localhost:3000/users \
-H "Content-Type: application/json" \
```

```
-d '{"name": "Akhil"}'
```

PUT (update user):

```
curl -X PUT http://localhost:3000/users/1 \  
-H "Content-Type: application/json" \  
-d '{"name": "Mohan Kumar"}'
```

DELETE a user:

```
curl -X DELETE http://localhost:3000/users/2
```

url Module in Node.js

What is the url module?

The `url` module helps you **parse**, **format**, and **resolve URLs**.

How to use it:

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

const parsed = url.parse('http://localhost:3000/users?id=101', true);
console.log(parsed);
```

Output Breakdown:

```
{  
  protocol: 'http:',  
  host: 'localhost:3000',  
  pathname: '/users',  
  query: { id: '101' }  
}
```

Common Methods

Method	Description
url.parse(url, true)	Parses a URL string into an object (true = parse query string too)
url.format(urlObj)	Turns a URL object back into a string
url.resolve(from, to)	Resolves relative URL against a base

Example: url.parse()

```
const parsedUrl =  
url.parse('http://localhost:3000/products?category=books&id=10',  
true);  
  
console.log(parsedUrl.pathname); // /products  
console.log(parsedUrl.query); // { category: 'books', id: '10' }  
console.log(parsedUrl.query.id); // 10
```

Summary Table

HTTP Methods

Method	Used For	Handled In <code>http.createServer()</code>
GET	Fetch data	<code>req.method === "GET"</code>
POST	Submit data	<code>req.method === "POST"</code>
PUT	Update data	<code>req.method === "PUT"</code>
DELETE	Delete data	<code>req.method === "DELETE"</code>

url Module Summary

Method	Description
<code>url.parse()</code>	Parses a URL into an object
<code>url.format()</code>	Builds a URL string from an object
<code>url.resolve()</code>	Resolves a relative path to full

4. NPM and Modules

What is NPM?

NPM stands for **Node Package Manager**.

It is the **default package manager for Node.js** and is used to:

- Install libraries & tools (`npm install`)
- Share reusable code
- Manage versions and dependencies
- Run scripts (`npm start`, `npm test`, etc.)

It connects to the <https://www.npmjs.com> registry which hosts thousands of open-source packages.

Types of Packages

Local Packages

- Installed in the current project directory
- Stored in `node_modules/`
- Listed in `package.json`

```
npm install express
```

Structure:

```
project/
└── node_modules/
    └── package.json
    └── app.js
```

Global Packages

- Installed system-wide
- Used via command-line tools

```
bash
CopyEdit
npm install -g nodemon
```

To check:

```
npm list -g --depth=0
```

Installing Third-party Packages

Example: Install express and nodemon

```
npm install express  
npm install --save-dev nodemon
```

package.json script for dev:

```
"scripts": {  
  "start": "node app.js",  
  "dev": "nodemon app.js"  
}
```

Now run:

```
npm run dev
```

Semantic Versioning (SemVer)

Used to manage version numbers like:

```
"express": "^4.17.1"
```

Format:

MAJOR.MINOR.PATCH

Segment	Meaning
---------	---------

MAJOR	Breaking changes (v2 → v3)
MINOR	New features (v2.1 → v2.2)
PATCH	Bug fixes (v2.1.0 → v2.1.1)

Symbols:

Symbol	Meaning
^	Accept upgrades within the same major version
~	Accept upgrades within the same minor version
No symbol	Lock to that exact version

5. Express.js (Node.js Framework)

What is Express?

Express.js is a **fast, minimal, and flexible web framework** for Node.js that simplifies:

- Handling routes (GET, POST, PUT, DELETE)
- Working with query/body/URL parameters
- Middleware (for authentication, logging, etc.)
- Error handling
- Serving static files
- Building APIs & web apps

Think of it as: “Node.js but with superpowers!”

How to Install:

`npm install express`

Or

```
npm i express
```

Creating a Simple Server

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

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

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

API Routes (GET, POST, PUT, DELETE)

Let's manage a list of employees.

1. GET – Fetch all employees

```
let employees = [
  { id: 1, name: 'Mohan' },
  { id: 2, name: 'Niranjan' }
];

app.get('/employees', (req, res) => {
  res.json(employees);
});
```

2. POST – Add a new employee

```
app.post('/employees', (req, res) => {
  const newEmployee = req.body;
  newEmployee.id = employees.length + 1;
  employees.push(newEmployee);
  res.status(201).json(newEmployee);
});
```

3. PUT – Update an employee

```
app.put('/employees/:id', (req, res) => {
  const id = parseInt(req.params.id);
  const updated = req.body;

  employees = employees.map(emp =>
    emp.id === id ? { ...emp, ...updated } : emp
  );

  res.json({ message: 'Employee updated' });
});
```

4. DELETE – Remove an employee

```
app.delete('/employees/:id', (req, res) => {
  const id = parseInt(req.params.id);
  employees = employees.filter(emp => emp.id !== id);
  res.json({ message: 'Employee deleted' });
});
```

Route Parameters vs Query Parameters

Route Parameters (:id)

Used to access dynamic segments in the URL.

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

URL:

/users/101

Output:

User ID: 101

Query Parameters (?key=value)

Used to send optional data.

```
app.get('/search', (req, res) => {
  res.send(`You searched for ${req.query.q}`);
});
```

URL:

/search?q=nodejs

Output:

```
You searched for nodejs
```

Middleware in Express

Middleware functions run **before the route is handled**.

Used for logging, authentication, body parsing, error handling, etc.

Example: Logger Middleware

```
app.use((req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next(); // pass to next middleware or route
});
```

Built-in Middleware

```
app.use(express.json()); // for parsing application/json
app.use(express.urlencoded({ extended: true })); // for form data
```

Error Handling Middleware

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).json({ error: 'Something went wrong!' });
});
```

Use `next(err)` to trigger it.

Full Example

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

let users = [{ id: 1, name: 'Mohan' }];

app.use(express.json());

app.get('/users', (req, res) => {
  res.json(users);
});

app.post('/users', (req, res) => {
  const user = req.body;
  user.id = users.length + 1;
  users.push(user);
  res.status(201).json(user);
});

app.put('/users/:id', (req, res) => {
  const id = parseInt(req.params.id);
  users = users.map(u => u.id === id ? { ...u, ...req.body } : u);
  res.json({ message: 'Updated' });
});

app.delete('/users/:id', (req, res) => {
  const id = parseInt(req.params.id);
  users = users.filter(u => u.id !== id);
  res.json({ message: 'Deleted' });
});

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

Summary Table

Concept	Description
express()	Create Express app
app.get()	Handle GET requests
app.post()	Handle POST requests
app.put()	Handle PUT requests (update)
app.delete()	Handle DELETE requests
req.params	Access route parameters (:id)
req.query	Access query strings (?q=value)
app.use()	Apply middleware globally
Error Middleware	Handle all application errors gracefully

What is MongoDB?

- MongoDB is a **NoSQL database** (not relational like MySQL).
- It stores data in **documents (JSON-like format)** instead of rows & tables.
- Each document is like a JavaScript object.
- Very flexible — you don't need to define columns before inserting data (schema-less).

Example MongoDB document in a users collection:

```
{  
  "_id": "64f3d3ab1234567890",  
  "name": "Mohan",
```

```
"age": 21,  
"skills": ["Node.js", "React", "MongoDB"]  
}
```

Collections = like **tables** in SQL

Documents = like **rows** in SQL

What is Mongoose?

- **Mongoose** is an **ODM (Object Data Modeling) library** for MongoDB in Node.js.
- It provides a **structured way** to define schemas, models, and interact with MongoDB.
- Without Mongoose, you talk to MongoDB directly (flexible but messy).
- With Mongoose, you define **schemas (rules)**, so your data stays consistent.

Example: Using MongoDB with Mongoose

1. Install

```
npm install mongoose
```

2. Connect to DB

```
const mongoose = require("mongoose");  
  
mongoose.connect("mongodb://127.0.0.1:27017/mydb")  
.then(() => console.log(" MongoDB Connected"))  
.catch(err => console.error(err));
```

3. Define Schema

```
const userSchema = new mongoose.Schema({  
  name: { type: String, required: true },
```

```
    age: Number,  
    skills: [String]  
});
```

4. Create Model

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

5. Use Model

```
// Insert user  
const newUser = new User({ name: "Mohan", age: 21, skills: ["Node.js",  
"React"] });  
await newUser.save();  
  
// Find users  
const users = await User.find();  
console.log(users);  
  
// Update user  
await User.updateOne({ name: "Mohan" }, { $set: { age: 22 } });  
  
// Delete user  
await User.deleteOne({ name: "Mohan" });
```

Difference Between MongoDB & Mongoose

Feature	MongoDB	Mongoose
Type	Database	ODM (library)
Schema	Schema-less (flexible)	Schema-based (structured)
Usage	Stores raw JSON docs	Adds models, validation, middlewares
Example	db.users.insert({name:"Mohan"})	new User({name:"Mohan"}).save()

Think of it like this:

- **MongoDB** = The database (where data lives).
- **Mongoose** = A smart middleman that makes working with MongoDB in Node.js cleaner & safer.

Folder Structure

```
project-root/
  -- server.js          # Entry point
  -- package.json

  -- config/            # Configuration (DB, env, etc.)
    -- db.js

  -- models/            # Mongoose schemas & models
    -- user.model.js

  -- controllers/       # Business logic (CRUD operations)
    -- user.controller.js

  -- routes/            # API routes
    -- user.routes.js

  -- middlewares/       # (Optional) middlewares (auth, error
    handling, etc.)
    -- errorHandler.js

  -- utils/             # Helper functions
    -- response.js
```

File Explanations

1. server.js (entry point)

```
const express = require("express");
const mongoose = require("mongoose");
const dotenv = require("dotenv");
const userRoutes = require("./routes/user.routes");

dotenv.config();
const app = express();

app.use(express.json());

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

// DB connection
mongoose.connect(process.env.MONGO_URI)
  .then(() => console.log(" MongoDB connected"))
  .catch(err => console.error("DB error:", err));

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

2. config/db.js

(If you want a separate file for DB connection)

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

const connectDB = async () => {
  try {
    await mongoose.connect(process.env.MONGO_URI);
    console.log(" MongoDB Connected");
  } catch (err) {
    console.error(" DB Connection Failed", err.message);
    process.exit(1);
  }
}
```

```
    }
};

module.exports = connectDB;
```

3. models/user.model.js

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

const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  age: { type: Number, required: true },
  email: { type: String, unique: true, required: true }
}, { timestamps: true });

module.exports = mongoose.model("User", userSchema);
```

4. controllers/user.controller.js

```
const User = require("../models/user.model");

// Create User
exports.createUser = async (req, res) => {
  try {
    const user = await User.create(req.body);
    res.status(201).json(user);
  } catch (err) {
    res.status(400).json({ error: err.message });
  }
};

// Get All Users
exports.getUsers = async (req, res) => {
  try {
    const users = await User.find();
```

```
    res.json(users);
} catch (err) {
    res.status(500).json({ error: err.message });
}
};

// Get Single User
exports.getUserById = async (req, res) => {
    try {
        const user = await User.findById(req.params.id);
        if (!user) return res.status(404).json({ message: "User not found" });
        res.json(user);
    } catch (err) {
        res.status(500).json({ error: err.message });
    }
};

// Update User
exports.updateUser = async (req, res) => {
    try {
        const user = await User.findByIdAndUpdate(req.params.id, req.body,
{ new: true });
        if (!user) return res.status(404).json({ message: "User not found" });
        res.json(user);
    } catch (err) {
        res.status(400).json({ error: err.message });
    }
};

// Delete User
exports.deleteUser = async (req, res) => {
    try {
        const user = await User.findByIdAndDelete(req.params.id);
        if (!user) return res.status(404).json({ message: "User not found" });
        res.json({ message: "User deleted successfully" });
    } catch (err) {
```

```
    res.status(500).json({ error: err.message });
}
};
```

5. routes/user.routes.js

```
const express = require("express");
const router = express.Router();
const userController = require("../controllers/user.controller");

// CRUD routes
router.post("/", userController.createUser);
router.get("/", userController.getUsers);
router.get("/:id", userController.getUserById);
router.put("/:id", userController.updateUser);
router.delete("/:id", userController.deleteUser);

module.exports = router;
```

API Endpoints

- POST /api/users → Create user
- GET /api/users → Get all users
- GET /api/users/:id → Get single user
- PUT /api/users/:id → Update user
- DELETE /api/users/:id → Delete user

INSERT METHODS (Create)

In Mongoose, inserting means adding documents into a collection.

1. Model.create(doc)

Creates and saves a new document directly.

```
const user = await User.create({ name: "Mohan", age: 21 });
```

2. new Model(doc).save()

Gives more control, lets you modify before saving.

```
const user = new User({ name: "Niranjan", age: 22 });
await user.save();
```

3. Model.insertMany([docs])

Insert multiple documents at once (faster).

```
await User.insertMany([
  { name: "Ravi", age: 25 },
  { name: "Prasad", age: 30 }
]);
```

FIND METHODS (Read)

Used to fetch data from MongoDB.

1. Model.find(query)

Finds **all** documents matching query.

```
const users = await User.find({ age: { $gte: 20 } });
```

2. Model.findOne(query)

Finds the **first match**.

```
const user = await User.findOne({ name: "Mohan" });
```

3. Model.findById(id)

Finds by `_id` directly.

```
const user = await User.findById("64f3d3ab1234567890");
```

UPDATE METHODS

Modify existing documents.

1. Model.updateOne(filter, update)

Updates **first matched document**.

```
await User.updateOne({ name: "Mohan" }, { $set: { age: 23 } });
```

2. Model.updateMany(filter, update)

Updates **all documents** matching filter.

```
await User.updateMany({}, { $set: { active: true } });
```

3. Model.findByIdAndUpdate(id, update, { new: true })

Finds by id and updates, returns updated doc if {new:true}.

```
const updated = await User.findByIdAndUpdate("64f3...", { $inc: { age: 1 } }, { new: true });
```

4. Model.findOneAndUpdate(filter, update, options)

Finds one and updates.

```
const user = await User.findOneAndUpdate(
  { name: "Niranjan" },
  { $set: { age: 25 } },
  { new: true }
);
```

DELETE METHODS

Remove documents.

1. Model.deleteOne(filter)

Deletes the **first matched**.

```
await User.deleteOne({ name: "Mohan" });
```

2. Model.deleteMany(filter)

Deletes **all matches**.

```
await User.deleteMany({ active: false });
```

3. Model.findByIdAndDelete(id)

Deletes by id.

```
await User.findByIdAndDelete("64f3...");
```

4. Model.findOneAndDelete(filter)

Deletes first match and returns deleted doc.

```
await User.findOneAndDelete({ name: "Ravi" });
```

COMMON MONGOOSE OPERATORS

These operators work inside queries and updates.

Comparison Operators

- \$eq → equal
- \$ne → not equal
- \$gt → greater than
- \$gte → greater than or equal
- \$lt → less than
- \$lte → less than or equal
- \$in → in array
- \$nin → not in array

```
await User.find({ age: { $gte: 20, $lte: 30 } });
await User.find({ name: { $in: ["Mohan", "Ravi"] } });
```

Logical Operators

- \$and, \$or, \$not, \$nor

```
await User.find({ $or: [{ age: { $lt: 18 } }, { age: { $gt: 60 } }] });
```

Update Operators

- **\$set** → set new value
- **\$unset** → remove a field
- **\$inc** → increment value
- **\$push** → add to array
- **\$pull** → remove from array
- **\$addToSet** → add to array if not already present

```
await User.updateOne({ name: "Mohan" }, { $inc: { age: 1 } });
await User.updateOne({ name: "Ravi" }, { $push: { skills: "Node.js" } });
```

Projection (selecting fields)

```
await User.find({}, "name age"); // only name and age
await User.find({}, { email: 0 }); // exclude email
```

Summary

- **Insert** → create, save, insertMany
- **Find** → find, findOne, findById
- **Update** → updateOne, updateMany, findByIdAndUpdate
- **Delete** → deleteOne, deleteMany, findByIdAndDelete
- **Operators** → \$gt, \$lt, \$set, \$inc, \$push, \$pull, \$or, \$and, \$in, \$nin

Common Mongoose Validations

1. Required

```
const userSchema = new mongoose.Schema({  
  name: { type: String, required: true }, // must be present  
});
```

2. String Validations

```
const userSchema = new mongoose.Schema({  
  username: {  
    type: String,  
    required: [true, "Username is required"],  
    minlength: [3, "Minimum 3 characters"],  
    maxlength: [15, "Maximum 15 characters"],  
    match: [/^[a-zA-Z0-9]+$/, "Only alphanumeric allowed"]  
  }  
});
```

3. Number Validations

```
const userSchema = new mongoose.Schema({  
  age: {  
    type: Number,  
    min: [18, "Age must be at least 18"],  
    max: [60, "Age must not exceed 60"],  
  }  
});
```

4. Custom Validation

```
const userSchema = new mongoose.Schema({
  email: {
    type: String,
    required: true,
    validate: {
      validator: function (value) {
        return /^[^_.+@.+\..+$/ .test(value); // simple regex
      },
      message: props => `${props.value} is not a valid email!`
    }
  }
});
```

5. Enum (Predefined Values Only)

```
const userSchema = new mongoose.Schema({
  role: {
    type: String,
    enum: ["admin", "user", "manager"],
    default: "user"
  }
});
```

6. Unique (not a true validation, but creates an index)

```
const userSchema = new mongoose.Schema({
  email: {
    type: String,
    unique: true, // creates a unique index in MongoDB
    required: true
  }
});
```

Example: Full Schema with Validations

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

const userSchema = new mongoose.Schema({
  name: {
    type: String,
    required: [true, "Name is required"],
    minlength: [3, "Name must be at least 3 characters"]
  },
  email: {
    type: String,
    required: [true, "Email is required"],
    unique: true,
    match: [/^.{+}@.{+}\..+$/, "Please enter a valid email"]
  },
  age: {
    type: Number,
    min: [18, "Age must be at least 18"],
    max: [60, "Age must not exceed 60"]
  },
  role: {
    type: String,
    enum: ["admin", "user", "manager"],
    default: "user"
  }
});

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

module.exports = User;
```

Example Usage

```
const User = require('./models/User');

// Try creating an invalid user
const user = new User({
  name: "Al", // too short
  email: "not-an-email", // invalid
  age: 15 // too low
});

user.save()
  .then(() => console.log("User saved!"))
  .catch(err => console.log("Validation Error:", err.message));
```

Output:

```
Validation Error: User validation failed:
name: Name must be at least 3 characters,
email: Please enter a valid email,
age: Age must be at least 18
```

Validation on Update

User Schema with Validations

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

const userSchema = new mongoose.Schema({
  name: {
    type: String,
    required: [true, "Name is required"],
    minlength: [3, "Name must be at least 3 characters"]
  },
  age: {
```

```
    type: Number,  
    min: [18, "Age must be at least 18"],  
    max: [60, "Age must not exceed 60"]  
  }  
});  
  
const User = mongoose.model("User", userSchema);  
module.exports = User;
```

Update Without Validation (default)

```
await User.findByIdAndUpdate("USER_ID", { age: 10 });
```

This will succeed, even though age is below 18 ✗.

Update With Validation

```
await User.findByIdAndUpdate(  
  "USER_ID",  
  { age: 10 },  
  { runValidators: true } // forces schema validation  
);
```

This will fail with:

```
ValidationError: Age must be at least 18
```

Options You Should Know

- `runValidators: true` → ensures schema validations run.
- `context: 'query'` → sometimes needed when using custom validators that depend on this.

Example:

```
await User.updateOne(  
  { _id: "USER_ID" },  
  { name: "Al" },  
  { runValidators: true, context: 'query' }  
);
```

Full Example with Try/Catch

```
try {  
  await User.findByIdAndUpdate(  
    "USER_ID",  
    { name: "Al", age: 15 },  
    { runValidators: true }  
  );  
} catch (err) {  
  console.log("Validation Error:", err.message);  
}
```

Output:

```
Validation Error: User validation failed:  
name: Name must be at least 3 characters,  
age: Age must be at least 18
```

Summary

- `.save()` & `.create()` → validations run automatically.
- `updateOne`, `findByIdAndUpdate`, `updateMany` → need `{ runValidators: true }`.
- Use `context: 'query'` if custom validators reference `this`.

Why context: 'query'?

In Mongoose, **custom validators** sometimes rely on this.

- In `.save()`, `this` refers to the **document**.
- In `.updateOne()` or `.findByIdAndUpdate()`, `this` refers to the **query**.

So, if you use a custom validator that checks `this`, you need to pass context: 'query' to make it work.

Example Schema with a Custom Validator

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

const userSchema = new mongoose.Schema({
  username: {
    type: String,
    required: true,
    minlength: [3, "Username must be at least 3 chars"],
  },
  email: {
    type: String,
    required: true,
    validate: {
      validator: function (value) {
        // Show what "this" points to
        console.log("Inside validator, this =", this);
      }
    }
  }
});
```

```
        return /^.+@.+\..+$/.test(value); // simple email check
    },
    message: (props) => `${props.value} is not a valid email!`,
},
},
});

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

Update with { runValidators: true } only

```
await User.updateOne(
  { _id: "USER_ID" },
  { email: "invalidemail" }, // invalid email
  { runValidators: true }
);
```

Output:

```
Inside validator, this = {}
ValidationError: users validation failed: email: invalidemail is not a
valid email!
```

Notice that this is {} (empty) because by default validators don't know about the query.

Update with { runValidators: true, context: 'query' }

```
await User.updateOne(
  { _id: "USER_ID" },
  { email: "invalidemail" },
  { runValidators: true, context: 'query' }
```

```
);
```

Output:

```
Inside validator, this = Query { ... }
ValidationError: users validation failed: email: invalidemail is not a
valid email!
```

Now this points to the **query object**, which you can use in advanced validators (like checking other fields in the same query).

Summary

- `runValidators: true` → enables schema validations on updates.
- `context: 'query'` → ensures `this` inside custom validators refers to the **query**, not an empty object.
- This is useful for **complex validations** (e.g., checking uniqueness with respect to another field).

Example: Create a User with Validations

1. Define Schema

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

const userSchema = new mongoose.Schema({
  name: {
    type: String,
    required: [true, "Name is required"],
    minlength: [3, "Name must be at least 3 characters"],
  },
  email: {
```

```

    type: String,
    required: [true, "Email is required"],
    unique: true,
    match: [/^.+@.+\\..+$/, "Please enter a valid email"],
  },
  age: {
    type: Number,
    min: [18, "Age must be at least 18"],
    max: [60, "Age must not exceed 60"],
  },
);
};

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

```

2. Create a Document

```

const mongoose = require("mongoose");
const User = require("./User"); // your schema file

mongoose.connect("mongodb://127.0.0.1:27017/testDB")
  .then(() => console.log("MongoDB connected"))
  .catch(err => console.error(err));

// Example: Creating a user
async function createUser() {
  try {
    const user = await User.create({
      name: "Al", // ✗ too short
      email: "not-an-email", // ✗ invalid
      age: 15 // ✗ below min
    });

    console.log("User created:", user);
  } catch (err) {
    console.log("Validation Error:", err.message);
  }
}

```

```
}

createUser();
```

3. Expected Output (Validation Error)

Validation Error: User validation failed:
name: Name must be at least 3 characters,
email: Please enter a valid email,
age: Age must be at least 18

4. Valid Data Example

```
await User.create({
  name: "Alice",
  email: "alice@gmail.com",
  age: 25
});
```

This will insert the document successfully.

Key Points

- `create()` and `save()` **always run validations automatically**.
- If validation fails → `ValidationError`.
- If all fields pass rules → document gets inserted.

⚡ So:

- For **creating documents**, validations run automatically.
- For **updating documents**, you must add `{ runValidators: true }`.

MongoDB + Mongoose: Associations (Relationships).

In MongoDB, there are **two ways** to create associations:

1. **Embedded Documents** (denormalization) → store related data inside the same document.
2. **Referenced Documents** (normalization) → store related data in separate collections and use `_id` references.

1. One-to-One (1:1)

Embedded

Definition: One document contains exactly one related document inside it.

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

const profileSchema = new mongoose.Schema({
  bio: String,
  website: String
});

const userSchema = new mongoose.Schema({
  name: String,
  profile: profileSchema // embedded
});

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

async function run() {
  await mongoose.connect("mongodb://127.0.0.1:27017/associations");
```

```

const user = new User({
  name: "Alice",
  profile: { bio: "Software Engineer", website: "alice.com" }
});

await user.save();
console.log(await User.find());
}

run();

```

Output

```
[
  {
    "_id": "650...",
    "name": "Alice",
    "profile": { "bio": "Software Engineer", "website": "alice.com" }
  }
]
```

Referenced

👉 **Definition:** Store related doc separately and reference it with `_id`.

```

const profileSchema = new mongoose.Schema({
  bio: String,
  website: String
});
const Profile = mongoose.model("Profile", profileSchema);

const userSchema = new mongoose.Schema({
  name: String,
  profile: { type: mongoose.Schema.Types.ObjectId, ref: "Profile" } // reference
});
```

```

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

async function run() {
  await mongoose.connect("mongodb://127.0.0.1:27017/associations");

  const profile = await Profile.create({ bio: "Engineer", website: "bob.com" });
  const user = await User.create({ name: "Bob", profile: profile._id });

  const result = await User.find().populate("profile");
  console.log(result);
}

run();

```

Output

```
[
  {
    "_id": "651...",
    "name": "Bob",
    "profile": { "_id": "652...", "bio": "Engineer", "website": "bob.com" }
  }
]
```

2. One-to-Many (1:N)

Embedded

A parent doc contains an **array of sub-documents**.

```

const postSchema = new mongoose.Schema({
  title: String,
  comments: [{ body: String, author: String }] // embedded array

```

```

});
```

```

const Post = mongoose.model("Post", postSchema);
```

```

async function run() {
  await mongoose.connect("mongodb://127.0.0.1:27017/associations");

  const post = new Post({
    title: "My First Post",
    comments: [
      { body: "Nice post!", author: "Alice" },
      { body: "Thanks for sharing", author: "Bob" }
    ]
  });

  await post.save();
  console.log(await Post.find());
}
```

```

run();
```

Output

```
[
{
  "_id": "653...",
  "title": "My First Post",
  "comments": [
    { "body": "Nice post!", "author": "Alice" },
    { "body": "Thanks for sharing", "author": "Bob" }
  ]
}]
```

Referenced

Store children in a separate collection, parent stores an **array of ObjectIds**.

```

const commentSchema = new mongoose.Schema({
  body: String,
  author: String
});
const Comment = mongoose.model("Comment", commentSchema);

const postSchema = new mongoose.Schema({
  title: String,
  comments: [{ type: mongoose.Schema.Types.ObjectId, ref: "Comment" }]
});
const Post = mongoose.model("Post", postSchema);

async function run() {
  await mongoose.connect("mongodb://127.0.0.1:27017/associations");

  const c1 = await Comment.create({ body: "Great!", author: "Alice" });
  const c2 = await Comment.create({ body: "Wow!", author: "Bob" });

  const post = await Post.create({ title: "Referenced Post", comments: [c1._id, c2._id] });

  const result = await Post.find().populate("comments");
  console.log(result);
}

run();

```

Output

```
[
  {
    "_id": "654...",
    "title": "Referenced Post",
    "comments": [
      { "_id": "655...", "body": "Great!", "author": "Alice" },
      { "_id": "656...", "body": "Wow!", "author": "Bob" }
    ]
  }
]
```

]

3. Many-to-Many (M:N)

Example: Students & Courses

- A student can join **many courses**.
- A course can have **many students**.

Referenced (most common way)

```
const studentSchema = new mongoose.Schema({
  name: String,
  courses: [{ type: mongoose.Schema.Types.ObjectId, ref: "Course" }]
});
const Student = mongoose.model("Student", studentSchema);

const courseSchema = new mongoose.Schema({
  title: String,
  students: [{ type: mongoose.Schema.Types.ObjectId, ref: "Student" }]
});
const Course = mongoose.model("Course", courseSchema);

async function run() {
  await mongoose.connect("mongodb://127.0.0.1:27017/associations");

  const s1 = await Student.create({ name: "Mohan" });
  const s2 = await Student.create({ name: "Prathyu" });

  const c1 = await Course.create({ title: "Math" });
  const c2 = await Course.create({ title: "Science" });

  // Link them (many-to-many)
  s1.courses.push(c1._id, c2._id);
  s2.courses.push(c1._id);
```

```

    await s1.save();
    await s2.save();

    c1.students.push(s1._id, s2._id);
    c2.students.push(s1._id);

    await c1.save();
    await c2.save();

    const result = await Student.find().populate("courses");
    console.log(result);
}
run();

```

Output

```
[
  {
    "_id": "657...",
    "name": "Mohan",
    "courses": [
      { "_id": "658...", "title": "Math" },
      { "_id": "659...", "title": "Science" }
    ]
  },
  {
    "_id": "660...",
    "name": "Prathyu",
    "courses": [
      { "_id": "658...", "title": "Math" }
    ]
  }
]
```

Summary

- **One-to-One**
 - Embedded → put profile inside user.
 - Referenced → user has profileId.
- **One-to-Many**
 - Embedded → parent stores array of children inside it.
 - Referenced → parent stores array of ObjectIds of children.
- **Many-to-Many**
 - Both collections keep array of ObjectIds referencing each other.