Section1: Introduction to Node.js and npm

1.1 What is Node.js? Breaking the JavaScript Mold

For the longest time, JavaScript lived only in the browser. It was a language designed to make web pages interactive. You could handle clicks, validate forms, and update content dynamically. But what if you wanted to read a file from your computer, connect to a database, or create a web server? That was the domain of languages like PHP, Python, Java, or C#.

Then came Node.js.

Node.js is a runtime environment that allows you to run JavaScript code outside of a browser, on your server or local machine.

Let's break down that definition:

Runtime Environment: Think of it as an operating system for your JavaScript code. It provides all the necessary components (like a JavaScript engine, libraries for I/O, etc.) that the code needs to execute.

Runs JavaScript outside the browser: This is the magic. Node.js takes the V8 JavaScript engine from Google Chrome and embeds it in a C++ program, giving it superpowers like file system access and network connectivity.

The Big Idea: The Problem Node.js Solves

Traditional web servers handled multiple users by spinning up a new thread (or process) for each simultaneous connection. This can be resource-intensive. Node.js introduced a different model: Single-Threaded, Event-Driven, and Non-Blocking I/O.

Single-Threaded: It uses just one main thread to handle all incoming requests.

Event-Driven & Non-Blocking: Instead of waiting for a time-consuming task to finish (like reading a file or fetching data from a database), Node.js registers a callback and moves on to the next request. When the slow task is done, an event is emitted, and the callback function is executed.

Analogy: Imagine a single waiter (the thread) in a restaurant. Instead of taking an order and then standing in the kitchen waiting for the food to be cooked (blocking), the waiter takes an order, tells the kitchen to start cooking, and immediately goes to serve another table (non-blocking). When the food is ready, the kitchen rings a bell (event), and the waiter

comes back to deliver the meal (callback). This allows one waiter to serve many tables efficiently.

1.2 Why Use Node.js? The Key Benefits

So, why has Node.js become such a powerhouse for backend development?

JavaScript Everywhere: You can use the same language (JavaScript) on both the frontend and the backend. This simplifies development, allows code sharing, and means your development team only needs to master one language.

High Performance & Scalability: The non-blocking architecture is exceptionally efficient for I/O-heavy applications, meaning it can handle a large number of simultaneous connections (like for chat apps, live notifications, or APIs).

Fast & Lightweight: Built on the incredibly fast V8 engine, Node.js starts quickly and uses resources efficiently.

Huge Ecosystem: This leads us directly to our next topic, npm. Node.js has the largest ecosystem of open-source libraries in the world, meaning you rarely have to build anything from scratch.

Perfect Use Cases for Node.js:

REST APIs and Microservices

Real-Time Applications (like chats, live feeds, gaming)

Data Streaming Applications (like Netflix)

Server-Side Rendering for frontend frameworks (Next.js, Nuxt.js)

1.3 Introducing npm: The Node Package Manager

If Node.js is the engine, npm is the fuel and the entire toolbox.

npm is two things:

The world's largest software registry: A massive online database containing millions of packages (reusable code libraries) that you can download for free and use in your projects.

A command-line tool: The program you use on your computer to interact with this registry—to install, manage, and share these packages.

Why is npm a game-changer?

Imagine you need to add user authentication, connect to a MongoDB database, or validate a form. Instead of writing thousands of lines of code yourself, you can simply run a command like npm install passport to get a battle-tested, community-supported library that does it for you.

Package: A folder containing code (a library) that you can use in your project.

Dependency: A package that your project needs to run. (e.g., your project depends on the express package).

package.json file: The heart of any Node.js project. It's a manifest file that lists your project's metadata, scripts, and—most importantly—all its dependencies. It's like the "ingredients list" for your project.

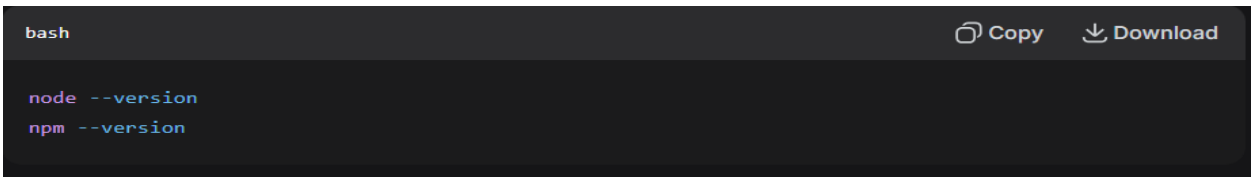
1.5 Your First Hands-On: Installing Node.js & npm

1. Go to nodejs.org

2. Download the LTS (Long-Term Support) version. Explain that LTS is the stable, recommended version for most users, while Current has the latest features but might be less stable

3. Run the installer. Walk them through the steps (it's typically just "Next, Next, Finish").

4. Verify the Installation.
Open your terminal (Command Prompt, PowerShell, or Terminal) and type:



You should see version numbers printed out. Congratulations! You now have both Node.js and npm installed on your machine

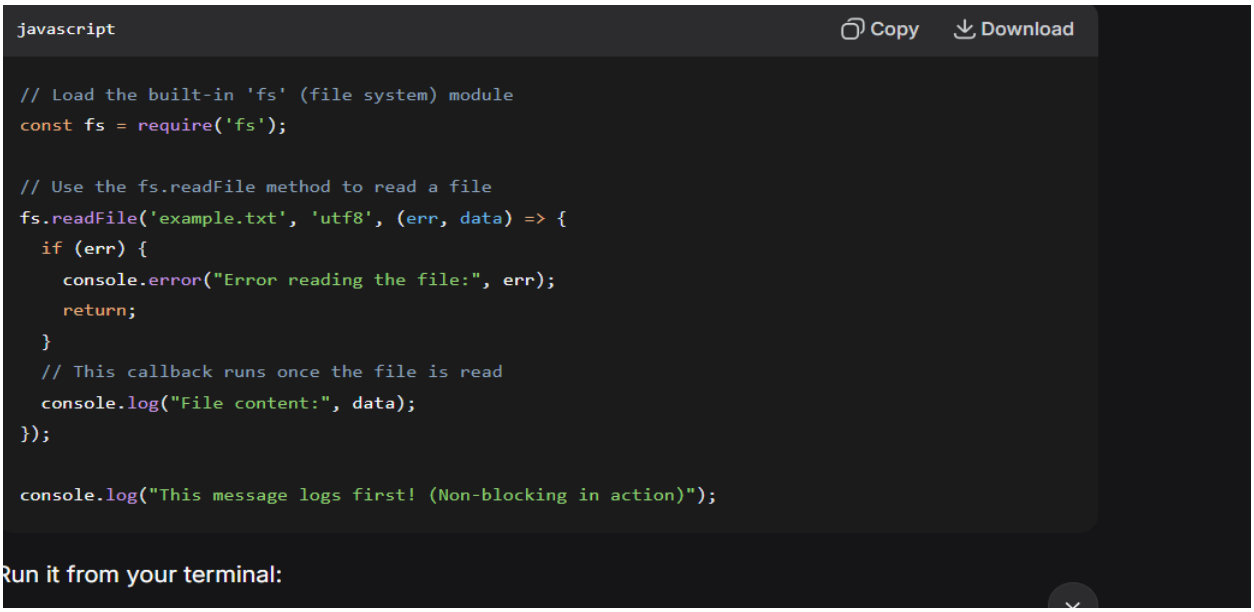
1.6 Core Module System: Node.js's Built-in Toolbox

Before we start installing external packages, it's important to know that Node.js comes with a powerful set of built-in modules for essential tasks. You don't need to install them; you just need to require them.

Example: **The fs (File System) Module**
Let's write a tiny script to prove Node.js can run outside the browser and read a file.

Create a file called app.js.

Write the following code:



Explanation: This demonstrates the non-blocking nature. The console.log at the end runs immediately, even while the file is being read. When the file reading is complete, the callback function is executed.

Other key built-in modules include http (to create servers!), path (for handling file paths), and os (for operating system info).

Section 2: Introduction to Backend Development

2.1: An Overview of How the Web Works

Key Concept: The Client-Server Model

Client: The user's device (browser, mobile app) that requests information

Server: A powerful computer that serves information

The Internet: The network that connects them

The Request-Response Cycle (Step by Step):

User Action: You type a URL (e.g., https://www.example.com) and hit Enter

DNS Lookup: Browser finds the server's actual address (IP address)

HTTP Request: Browser sends a request to that server: "Hey, give me the homepage!"

Server Processing: Backend application receives the request, may:

- Talk to a database
- Process business logic
- Gather data from other services

HTTP Response: Server sends back:

- Status Code (200 OK, 404 Not Found)
- Headers (metadata about the response)
- Body (the actual HTML/JSON/data)

Browser Rendering: Client receives response and displays the webpage

Analogy: Ordering food delivery

You (Client) → Place order (Request)

Restaurant (Server) → Prepares food (Processing)

Delivery driver (Network) → Brings food (Response)

2.2: HTTP Actions (Methods) - The Language of Web Communication

HTTP Methods = Verbs that define the action to be performed

GET,POST,PUT/PUTCH,DELETE

HTTP Status Codes - The Server's Response Language:

200 OK - Success!

201 Created - Resource successfully created

400 Bad Request - Client made an error

404 Not Found - Resource doesn't exist

500 Internal Server Error - Server problems

Frontend vs Backend - The Complete Division

Section 3: How Node.js Works - Behind the Scenes

3.1: Node.js Architecture: The Big Picture

Node.js = V8 JavaScript Engine + libuv + Other C++ Additions

1. V8 JavaScript Engine:

Developed by Google for Chrome

Compiles JavaScript to machine code (not interpretation)

Responsible for executing your JavaScript code

2. libuv Library:

Cross-platform C library

Provides the Event Loop and Thread Pool

Handles asynchronous I/O operations (file system, networking)

3. C++ Bindings:

   Allow JavaScript to communicate with low-level C/C++ code

4. JavaScript Core Library:

   The built-in modules we use (fs, http, path, etc.)

5. Key Insight:

Node.js isn't just JavaScript - it's a sophisticated runtime built on top of powerful C/C++ components.

3.2: The Event Loop - Heart of Node.js

What is the Event Loop?

A single-threaded loop that coordinates and dispatches operations

Constantly checks: "Is there any work to do?"

Runs in the same thread as your JavaScript code

How It Works:

Checks for pending timers (setTimeout, setInterval)

Checks pending I/O operations (file system, network requests)

Polls for new I/O events

Checks setImmediate callbacks

Handles 'close' events

Repeat!

```
console.log('Start');


setTimeout(() => console.log('Timeout 1'), 0);

setImmediate(() => console.log('Immediate 1'));

Promise.resolve().then(() => console.log('Promise 1'));


console.log('End');


// Output order explanation:

// 'Start'

// 'End'

// 'Promise 1' (Microtask queue)

// 'Timeout 1' (Timer queue)

// 'Immediate 1' (Check queue)
```

3.3 Single-Threaded but Non-Blocking: The Magic Explained

Myth Busting: "Node.js is single-threaded, so it can't handle multiple tasks"

Reality: Node.js uses a single main thread for JavaScript execution, but delegates I/O operations to system kernels or the thread pool.

Blocking vs Non-Blocking Code:

Section4: Introduction MongoDB

4.1 What is MongoDB?

MongoDB is an open-source, NoSQL database designed for storing and managing large volumes of unstructured or semi-structured data.
Instead of tables and rows (like SQL), it stores data in JSON-like documents inside collections.

Why MongoDB?

MongoDB was created to solve the limitations of traditional relational databases when handling:

- Big data
- Rapidly changing schemas
- Real-time applications
- Scalability needs

It provides flexibility, scalability, and high performance — making it ideal for modern web apps and APIs.

Key Idea:

- SQL → Tables → Rows
- MongoDB → Collections → Documents

4.2. SQL vs NoSQL (Comparison)

Feature	SQL Databases	MongoDB (NoSQL)
Data Format	Tables (rows & columns)	Collections (JSON-like documents)
Schema	Fixed	Dynamic / Flexible
Scalability	Vertical (add more power to one server)	Horizontal (add more servers easily)
Query Language	SQL	MongoDB Query Language (MQL)

Relationships	Relational (joins)	Embedded or referenced documents
Best For	Structured data	Unstructured or rapidly changing data

4.3. MongoDB Data Structure

Basic Terminology

Term	Description	Analogy (SQL)
Database	Container for collections	Database
Collection	Group of documents	Table
Document	Record stored in BSON (Binary JSON) format	Row
Field	Key-value pair inside a document	Column

Example document

```
{
  "_id": "67188a1f5a9d3",
  "name": "Firagos",
  "email": "jemal@example.com",
  "age": 24,
  "skills": ["Node.js", "React", "MongoDB"]
}
```

MongoDB Client (e.g. Node.js app) > MongoDB Server (Stores databases) > Collections & Docs

4.4. Installing MongoDB

Option 1: Local Installation

1. Visit: https://www.mongodb.com/try/download/community

2. Choose your OS and install.
3. Verify installation:
   mongo --version
   mongod –version
4. Start the MongoDB service (on Windows):
   net start MongoDB

4.5 Connecting Node.js with MongoDB

In your node js project install mongoose

npm install mongoose

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

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

Section5:Master Error Handling in Express.js - From Chaos to Control

Why Error Handling Matters

The Problem: Without proper error handling, your Express app CRASHES when errors occur. Users see broken connections, and your application becomes unreliable.

The Goal: Handle errors gracefully to provide:

Better user experience

Stable application that doesn't crash

Clear error messages for clients

Professional, production-ready apps

Two Types of Errors

1. Operational Errors (Expected Issues)

Invalid user input

Database connection failures

Missing resources (404)
Network issues
Solution: Handle gracefully with proper responses

2. Programmer Errors (Bugs)

undefined property access
Syntax errors
Async/await mistakes
Solution: Fix the code, but handle safely

Level 1: Basic Try-Catch

The WRONG Way (Dangerous)

```avascript

pp.post('/users', async (req, res) => {
 const newUser = await User.create(req.body); // CRASHES if error!
 res.status(201).json({ success: true, data: newUser });
);
```

Level 2: Using Next() for Error Propagation

Key Concept: Pass errors to Express's error handling chain using next(error)

```
app.get('/users/:id', async (req, res, next) => {
  try {
    const user = await User.findById(req.params.id);

    if (!user) {
      return next(new Error('User not found')); // Skip to error handler
    }

    res.status(200).json({ success: true, data: user });
  } catch (error) {
    next(error); // Pass async errors to central handler
  }
});
```

Level 3: Custom Error Classes

Create Custom Error Class (utils/ErrorResponse.js):

```
class ErrorResponse extends Error {
  constructor(message, statusCode) {
    super(message);
    this.statusCode = statusCode;
  }
}

module.exports = ErrorResponse;
```

Use Custom Errors in Routes:

```javascript
const ErrorResponse = require('../utils/ErrorResponse');

app.post('/login', async (req, res, next) => {
  const { email, password } = req.body;

  if (!email || !password) {
    return next(new ErrorResponse('Please provide email and password', 400));
  }

  const user = await User.findOne({ email });
  if (!user) {
    return next(new ErrorResponse('Invalid credentials', 401));
  }
});
```

Level 4: Global Error Handler Middleware

Create Error Handler (middleware/error.js):

```
const ErrorResponse = require('../utils/ErrorResponse');

const errorHandler = (err, req, res, next) => {
  let error = { ...err };
  error.message = err.message;

  // Log error for development
  console.log(err.stack.red);

  // Mongoose bad ObjectId
  if (err.name === 'CastError') {
    const message = 'Resource not found';
    error = new ErrorResponse(message, 404);
  }

  // Mongoose duplicate key
  if (err.code === 11000) {
    const message = 'Duplicate field value entered';
    error = new ErrorResponse(message, 400);
  }
```

```
  // Mongoose validation error
  if (err.name === 'ValidationError') {
    const message = Object.values(err.errors).map(val => val.message).join(', ');
    error = new ErrorResponse(message, 400);
  }

  // Send response to client
  res.status(error.statusCode || 500).json({
    success: false,
    error: error.message || 'Server Error'
  });
};

module.exports = errorHandler;
```

Use in Your App (app.js):

```
const errorHandler = require('./middleware/error');

// Your routes here...
app.use('/api/v1/users', userRoutes);

// ERROR HANDLER MUST BE LAST!
app.use(errorHandler);
```

Section6: Authentication & Authorization

Learning Objectives:

Understand difference between authentication and authorization

Implement JWT-based authentication

Create protected routes and role-based access

Build secure password handling

Key Definitions:

Authentication: "Who are you?"

Verifying user identity

Examples: Login, JWT tokens

Answer: "This user is John Doe"

Authorization: "What can you do?"

Checking user permissions

Examples: Admin routes, user roles

Answer: "John can delete posts but Jane cannot"

JWT (JSON Web Tokens) Explained:

Structure: Header.Payload.Signature

Payload: Contains user data (id, role)

Signature: Verifies token authenticity

Stateless: Server doesn't store session data

LESSON 2: USER MODEL & PASSWORD SECURITY

Password Security Fundamentals:

Never store plain text passwords

Use bcrypt for hashing with salt rounds

Auto-hash before saving to database

Never return password in API responses

User Model Key Features:

Email validation with regex

Role-based enum (user, publisher, admin)

Password field with select: false

- Automatic timestamping

LESSON 3: AUTHENTICATION FLOW

Registration Process:

- User provides name, email, password
- Validate input data
- Hash password automatically via middleware
- Create user in database
- Generate and return JWT token

Login Process:

- User provides email and password
- Find user by email including password field
- Compare provided password with stored hash
- If match, generate and return JWT
- If no match, return 401 error

Critical Security Notes:

- Always validate both email and password exist
- Use consistent error messages ("Invalid credentials")
- Include password with .select('+password') for comparison only
- Return same error for both wrong email and wrong password

LESSON 4: PROTECTING ROUTES

Middleware Concept:

- Functions that run before route handlers
- Can modify request/response objects
- Can end request-response cycle
- Call next() to continue to next middleware

Protect Middleware Flow:

- Check for token in Authorization header
- Verify token format: "Bearer <token>"
- Verify JWT signature and expiration
- Extract user ID from token payload
- Find user in database by ID
- Attach user to request object
- Call next() or return error

LESSON 5: AUTHORIZATION & ROLES

Role-Based Access Control:

- Different users have different permissions
- Pre-defined roles (user, publisher, admin)
- Middleware checks user role against required roles

Authorization Middleware:

- Higher-order function that returns middleware
- Takes allowed roles as parameters
- Checks if req.user.role is in allowed roles
- Returns 403 if user lacks permission

LESSON 6: SECURITY BEST PRACTICES

Essential Security Measures:

- Environment Variables: Never hardcode secrets
- Password Hashing: Always use bcrypt with salt
- JWT Expiration: Set reasonable token lifetimes
- HTTP-only Cookies: For production token storage
- Input Validation: Validate all user input

Common Vulnerabilities to Avoid:

Plain text password storage

Detailed error messages revealing system information

Missing input validation

Inadequate password strength requirements

Long-lived tokens without expiration