NODE . JS

Node.js is a cross-platform runtime ~~application~~ environment and library for running JavaScript applications outside the browser. It is used for creating server-side and networking web applications. It is open source and free to use.

Many of the basic modules of Node.js are written in JavaScript. Node.js is mostly use to run real-time server applications.

Node.js also provides a rich library of various JavaScript modules to simplify the development of web applications.

Node.js = Runtime Environment + JavaScript library

Features of Node.js

Following is a list of some important features of Node.js that makes it the first choice of software architects.

1) Extremely fast
2) I/O is Asynchronous and Event Driven.
3) Single threaded
4) Highly scalable
5) No buffering
6) Open source
7) License

Role of Node.js in the MERN Stack

It serves as a backend engine (runtime environment)

It serves as —

1. Backend Server Creation → builds the server that handles requests and responses
2. Connects Frontend and Database → Node.js acts as a bridge between the react frontend (client) and the MongoDB database (data layer)

   It uses Express.js and Mongoose to receive data from React, store or fetch data from MongoDB and send results back to React.
3. Hosts the Express Framework → Express.js runs on top of Node.js.
4. Handles Asynchronous Operations
5. Uses npm → provides access to thousands of ready-made modules.

Simple Flow Diagram

[React Frontend]

↓

(Sends API Request)

↓

[Node.js + Express Backend]

↓

[MongoDB Database]

↓

(Returns Response)

@ SYNTAX ERROR
— Abhishek Rathor

Main Components of Node.js

1. V8 JavaScript Engine

→ developed by Google written in C++
→ It converts JavaScript code into machine code directly.
→ ensures high performance and fast execution
→ The same engine used in the Google Chrome browser.

Role in Node.js
Executes JavaScript code efficiently on the server side.

Example :

```
console.log("Executed by V8 Engine");
```

2. Libuv (Library for Asynchronous I/O)

- A C library that provides Node.js with :
  - Event loop
  - Asynchronous I/O operations
  - Thread pool.

It helps Node.js handle :
- File operations
- Network requests
- Timers
  ↳ without blocking the main thread.

3. Node.js APIs (Bindings & Core Modules)

Node.js includes built-in APIs that developers can use directly — no need for external libraries.

Common Core Modules:

Module	Purpose
http	→ create web servers
fs	→ file system operations (read/write files)
path	→ handle file and directory paths
os	→ system info (CPU, memory, OS details)
url	→ parse and format URLs
events	→ handle custom events
crypto	→ hashing, encryption, security.

In short

Component	Description	Function
V8 Engine	Google's JS Engine	Executes JS code.
Libuv	C library	Handles async I/O operation
Core Modules	Built-in Node.js libraries	Enable system and server tasks.

npm (Node Package Manager)

What is npm?

npm stands for Node Package Manager. It is default package manager for Node.js and is used to :

- Install, manage, and update libraries or dependencies.
- Share and reuse code packages across projects

Every MERN project uses npm to manage backend and frontend dependencies.

Why npm is Important in the MERN Stack

The MERN stack depends on several Node.js packages such as :

- express → Backend framework
- mongoose → MongoDB integration
- cors, dotenv, bcrypt, jsonwebtoken → Security & configuration
- nodemon → Development tool

All of these are installed and managed using npm.

How npm works

When you install Node.js, npm gets installed automatically.

It provides :

1. Command Line Tool (CLI) → For installing and manging packages.
2. Online Registry → Stores thousands of open-source packages.

Example :

```
npm install express
```

npm Packages

A package (or module) is a collection of JavaScript files bundled together to perform specific functionality.

Examples:

Package	Purpose
express	→ backend framework for APIs
mongoose	→ MongoDB connection and schema modeling
cors	→ allows cross-origin requests (React → Node)
dotenv	→ loads environment variables
bcrypt	→ password hashing
jsonwebtoken	→ user authentication
nodemon	→ automatically restarts server when files change

Common npm Commands

Command	Description
npm init	→ Initializes a new Node.js project (creates package.json)
npm install <package>	→ Installs a specific package locally
npm install -g <package>	→ Installs a package globally
npm install	→ Installs all dependencies listed in package.json
npm uninstall <package>	→ Removes a package

npm update <package> → updates a package to latest version
npm list → lists all installed packages
npm start → runs the start script from package.json
npm run <script> → runs a custom script defined in package.json

package.json File

Every Node.js or MERN project includes a package.json file.

Purpose:
- Stores project info (name, version, description)
- Lists dependencies (required packages)
- contains scripts for automation

Example:

```
{
  "name": "mern-backend",
  "version": "1:0.0",
  "description": "Backend for MERN project",
  "main": "server.js",
  "scripts": {
      "start": "node server.js",
      "dev": "nodemon server.js"
  },
  "dependencies": {
      "express": "^4.18.2",
      "mongoose": "^7.0.3",
      "dotenv": "^16.0.0"
  }
}
```

package-lock.json

- Automatically created when you install packages.
- Records exact versions of dependencies and sub-dependencies.
- ensures the same versions are installed when sharing the project.

node_modules Folder

- stores all installed npm packages
- created automatically after running npm install.
- should not be uploaded to Github (it's large) → instead, share package.json.

Local vs Global Installation

Type	Command Example	Description
Local	npm install express	Installs inside current project (recommended)
Global	npm install -g nodemon	Available system-wide (for tools like nodemon)

Scripts in npm

You can define custom scripts in package.json and run them using npm run.

Example:

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

npm in the MERN Development workflow

Stack Layer	Example npm Packages
Backend (Node + Express)	express, cors, dotenv, bcrypt, jsonwebtoken
Database (MongoDB)	mongoose
Frontend (React)	react, react-dom, axios
Development Tools	nodemon, concurrently

Example: Installing All Dependencies in a MERN Project

```
npm install express mongoose dotenv cors
  bcrypt jsonwebtoken
npm install --save-dev nodemon
```

Then add this in package.json:

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

Now run the backend in development mode:

```
npm run dev
```

Asynchronous Programming in Node.js

Node.js is asynchronous and non-blocking, which means it can handle multiple operations at the same time without waiting for one to finish before starting another.

- In synchronous programming, tasks run one after another – each must finish before the next starts.
- In asynchronous programming, tasks can start, continue, and complete independently.
- Node.js uses an event loop to manage async data / tasks, allowing it to handle many requests efficiently.

Example :

```
console.log ("Start");
setTimeout (() => {
        console.log ("Async Task Done."); }
    2000);
console.log ("End");
```

Output :

```
Start
End
Async Task Done
```

"Async Task Done" runs later because setTimeout() is non-blocking.

Event-Driven Programming

Node.js uses an EventEmitter model - it listens for events and triggers callbacks (listeners) when those events occur.

Example:

```
const EventEmitter = require('events');
const emitter = new EventEmitter();
emitter.on('start', () => {
    console.log('Started!');  });
emitter.emit('start');
```

used in: HTTP servers, streams, file handling, etc.

Error Handling in Async Code

Always handle errors in asynchronous features using try/catch (for asyn/await) or .catch() (for promises)

Example with Async/Await

```
async function processData() {
  try {
    let result = await fetchData();
    console.log(result); }
  catch (error) {
    console.error("Something went wrong:", error); } }
```

Example with Promise:

```
fetchData()
  .then(result => console.log(result))
  .catch(err => console.error(err));
```

Summary Table

Concept	Description	Example
Callback	Function passed to another function	fs.readFile('file.txt', cb)
Promise	Object representing eventual completion/failure.	.then()..catch()
Async/Await	syntactic sugar over Promises	await fetch()
Event-driven	Responds to emitted events	emitter.on()
Error handling	Use try/catch or .catch()	try {...} catch(e) {}

Working with Databases (MongoDB Integration)

Node.js applications often use MongoDB as a NoSQL database. The connection between Node.js and MongoDB is usually handled through Mongoose, an object Data Modeling (ODM) library.

Connecting Node.js to MongoDB using Mongoose

Mongoose simplifies interaction with MongoDB by providing a scheme-based structure to your documents.

Installation `npm install mongoose`

Connection setup

```
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost:27017/myDatabase')
  .then(() => console.log("connected to MongoDB"))
  .catch(err => console.log("Connection failed:", err));
```

Note:

- `mongodb://localhost:27017` → Default MongoDB address
- `myDatabase` → your database name
- Then `.then()` and `.catch()` handle success/failure asynchronously.

Defining Schemas and Models

A Schema defines the structure of a document while a Model is used to interact with the database.

Schema Example

```
const mongoose = require('mongoose');
const userSchema = new mongoose.Schema({
    name: { type: String, required: true },
    age: Number, email: { type: String, unique: true,
required: true },
    createdAt: { type: Data, default: Date.now }
});
```

Creating a Model const User = mongoose.model('User', userSchema);

Schema → Model → Collection → Document

Schema (structure) → Model (logic) → Collection (users) → Document (one user)

Handling Async Queries (Async/Await)

Instead of using .then() and .catch(), you can use async/await for cleaner syntax.

Example

```
async function getUsers() {
  try {
    const users = await User.find();
    console.log(users);
  } catch (error) {
    console.log("Error fetching users:", error); }
}
getUsers();
```

Validation and Error Handling

Mongoose provides built-in validation for data consistency.

Example

```
const productSchema = new mongoose.Schema({
  name: { type: String, required: [true, "Product name required"] },
  price: { type: Number, min: [0, "Price must be positive"]
```

```
}, category : { typ: String, enum : ["Food", "Clothing", "Electronics"] }
});
```

Error handling

```
async function createProduct() {
  try {
    const Product = mongoose.model('Product', productSchema);
    const item = new Product({ price: -20 });
    await item.save();
  } catch (err) {
    console.error("Validation Error:", err.message);
} }
createProduct();
```

Mongoose Validation Features

- required, min, max, enum, match
- Custom validators using validate:
  { validator: fn, message: "..." }

Node.js ↔ Mongoose ↔ MongoDB

Client (React)
↓
Express.js Routes
↓
Mongoose Models
↓
MongoDB Collections

Environment Variables

Environment variables are key-value pairs used to store configuration data outside the code.
They help you secure sensitive information like API keys, database URIs, and passwords.

Example:

```
PORT = 5000
MONGO_URI = mongodb://localhost:27017/myDB
JWT_SECRET = mySuperSecretKey
```

Securing API Keys & Credentials

- Store keys, tokens, DB credentials inside .env files.
- Access using process.env.KEY_NAME.

Example

```
const apiKey =
    process.env.MY_API_KEY;
```

API Testing and Debugging

Testing with Postman or Thunder Client

- Test API endpoints (GET, POST, PUT, DELETE)
- Check headers, params, and JSON responses

```
}, category : { typ: String, enum : ["Food", "Clothing", "Electronics"]}
});
```

Error handling

```
async function createProduct() {
    try {
        const Product = mongoose.model('Product', productSchema);
        const item = new Product ({ price : -20 });
        await item.save();
    } catch (err) {
        console.error("Validation Error :", err.message);
} }
createProduct();
```

Mongoose Validation Features

- required, min, max, enum, match
- Custom validators using validate :
  { validator : fn, message : "..." }

Node.js ↔ Mongoose ↔ MongoDB

Client (React)
↓
Express.js Routes
↓
Mongoose Models
↓
MongoDB Collections

Using console.log() and morgan

npm install morgan

```
const morgan = require('morgan');
app.use(morgan('dev'));
```

Shows request logs in the console — method, URL, status, and time

Logging & Error Tracking

- Use console.error() for errors.
- In production, use logging libraries like winston or pino.

Deployment

Preparing for Production

- Remove console logs.
- Handle errors gracefully
- Use process.env.PORT.
- Use helmet for security headers.

Using Environment Variables for Production

```
NODE_ENV = production
PORT = 8080
MONGO_URI = "mongodb+srv://cluster.mongodb.net/myDB"
```

Express.JS

Express is a fast, assertive, essential and moderate web framework of Node.js. You can assume express as a layer built on the top of the Node.js that helps manage a server and routes. It provides a robust set of features to develop web and mobile applications.

Let's see some of the core features of Express framework:

- It can be used to design single-page, multi-page and hybrid web applications.
- It allows to setup middlewares to respond to HTTP Requests.
- It defines a routing table which is used to perform different actions based on HTTP method and URL.
- It allows to dynamically render HTML pages based on passing arguments to templates

Why use Express

- Ultra fast 1/10
- Asynchronous and single threaded
- MVC like structure
- Robust API makes routine easy

How does Express look like

Let's see a basic Express.js app

File : basic_express.js

```
var express = require ('express');
var app = express ();
app.get ('/', function (req, res) {
res. send ('Welcome to JavaTpoint'); });
```

```
var server = app.listen(8000, function () {
    var host = server.address().address
    var port = server.address().port
    console.log("Example app listening at https://%s:%s",
                host, port);
});
```

Setting Up Express

```
npm init -y
npm install express mongoose dotenv cors
```

Then create a file : server.js :

```
const express = require('express');
const app = express();
const PORT = 5000;
// Middleware
app.use(express.json()); // To parse JSON requests

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

// Start server
app.listen(PORT, () =>
    console.log('Server running on port ${PORT}'));
```

Now run :

```
node server.js
```

Basic Express Concepts

Each route in Express receives two main objects:

Express.js Request Object

The express.js request object represents the HTTP request and has properties for the request query, string, parameters, body, HTTP headers, and soon.

```
app.get('/', function (req, res) { //-- })
```

Properties

The following table specifies some of the properties associated with request object.

Object properties	Description
req.app	→ used to hold a reference to the instance of the express application i.e. using the middleware
req.body	→ contains key-value pairs of data submitted in the request body
req.ips	→ when trust proxy setting is true, this property contains an array of ip address specified in the ?x-forwarded-for? request header
req.params	→ an object containing properties mapped to the named route? parameters?
req.query	→ an object containing a property for each query string parameter in the route
req.route	→ the currently matched route a string
req.secure	→ a boolean that is true if a TLS connection is established

@ SYNTAX ERROR
— Abhishek Rathor

Express.js Response Object

The response object (res) specifies the HTTP response which is sent by an Express app when it gets an HTTP request.

What it does

- It sends response back to the client browser.
- It facilitates you to put new cookies value and that will write to the client browser (under cross domain rule)
- Once you res.send() or res.direct() or res.render(), you cannot do it again otherwise, there will be uncaught error.

Properties

Let's see some properties of response object

Object Properties	Description
res.app	→ It holds a reference to the instance of the express application that is using the middleware.
reshedders.sent	→ It is a Boolean property that indicates if the app sent HTTP headers for the response.
res.locals	→ It specifies an object that contains response local variables scoped to the request.

Example :

```
app.post ('/user', (req, res) => {
    const data = req.body; // get JSON data
res.status (201).json ({ message: 'User created', user : data });
});
```

Express.js Routing

Routing is made from the word route. It is used to determine the specific behavior of an application. It specifies how an application responds a client request to a particular route URL or path and a specific HTTP request method (GET, POST etc.). It can handle different types of HTTP requests.
Let's take an example to see basic routing.

Example :

```
app.get ('/users', (req, res) => {... }); // Read data
app.post ('/users', (req, res) => {... }); // Create new
app.put ('/users/:id', (req, res) => {... }). // update by Id
app.delete ('/users/:id', (req, res) => {... }); // Delete
```

Each route can also handle route params :

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

Express.js Middleware

Express.js Middleware are different types of functions that are involved in and invoked by the Express.js routing layer before the final request handler. As the name specified, middleware appears in the middle between an initial request and final intended route. In stack, middleware functions are always invoked in the order in which they are added.

Middleware is commonly used to perform tasks like body parsing for URL-encoded or JSON requests, cookie parsing for basic cookie handling or even building Javascript modules on the fly.

What is a Middleware function?

Middleware function are the functions that access to the request and response object (req, res) in request-response cycle.

A middleware function can perform the following tasks:

- It can execute any code.
- It can make changes to the request and the response objects
- It can end the request-response only cycle.
- It can all the next middleware function in the stack.

Express.js Middleware

Following is a list of possibly used middleware in Express.js app:

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

Common uses:

- Parse JSON
- Enable CORS
- Handle authentication
- Log requests

Example:

```
app.use(express.json()); // Built-in
app.use(cors()); // Cross-Origin Resource Sharing
```

Custom middleware:

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

Use of Express.js Middleware

If you want to record every time you get a request then you can use a middleware.

File: simple-middleware.js

```
var express = require('express')
var app = express();
app.use('function (req, res, next) {
    console.log('%s %s', req.method, req.url);
next();
});
app.get('/', function (req. res. next) {
    res.send('How can I help you?'); });
```

```
var server = app.listen (8000. function () {
    var host = server. address (). address
    var port = server. address (). port
    console. log ("Example app listening at https: %s %os host. port)
})
```

You see that server is listening
Now, you can see the result generated by server on the local host http://127.0.0.1:8000

Output

sample app listening at http://:::8000
you can see that output is same but command prompt is displaying a GET result.

Go to http://127.0.0.1:8000/help

How can I help You?
As many times as you reload the page, the command prompt will be updated
Note: In the above example next() middleware is used.

Express.js Scaffolding

Scaffolding is a technique that is supported by some MVC frameworks. It is mainly supported by the following frameworks:
Ruby on Rais, OutSystem Platform, Express Framework, play Framework Django etc.

Scaffolding facilitates the programmers to specify how the application data may be used. This specification is used by the frameworks with predefined code templates to generate the final code that the application can use for CRUD operations.

Express.js Scaffold → An Express.js scaffold supports candy and more web projects based on Node.js.

Express + MongoDB (via Mongoose)

Connection setup:

```
const mongoose = require('mongoose');
require('dotenv').config();
mongoose.connect(process.env.MONGO_URI).then(()
  => console.log('MongoDB Connected'))
.catch(err => console.log(err));
```

Example Model:

```
const userSchema = new mongoose.Schema({
    name: String,
    email: String, });
const User = mongoose.model('User', userSchema);
```

Example Route:

```
app.post('/users', async (req, res) => {
try { const user = new User(req.body);
    const saved = await user.save();
res.status(201).json(saved); } catch (err) {
```

res.status(400).json({ error: err.message }); } });

RESTful APIs

Express is perfect for building REST APIs that React frontend can call.

Method	Meaning	Example URL
GET	Read data	/api/users
POST	Create data	/api/users
PUT	update data	/api/users/:id
DELETE	remove data	/api/users/:id

Example

```
app.get('/api/users', async (req, res)
=> {
    const users = await User.find();
    res.json(users);
});
```

Error Handling in Express.js

Error handling means catching and responding to any problems that occur while your backend processes a request — without crashing the server.

Types of Errors :

1. Synchronous errors

2. ~~Any~~ Asynchronous errors
3. Operational errors
4. Programming errors

Basic Error Handling

You can catch and respond with an error message instead of crashing.

```
app.get('/example', (req, res) => {
    try {
        // something goes wrong
        throw new Error('Oops! Something went wrong.');
    } catch (err) {
        res.status(500).json({ message: err.message }); } });
```

Output : {"message": "Oops! Something went wrong."}

Centralized Error Handling Middleware

Create a custom error middleware :

```
// middleware/errorMiddleware.js
const errorHandler = (err, req, res, next) => {
    console.error(err.stack);
    res.status(err.statusCode || 500).json({
        success: false
        message: err.message || 'Internal Server Error' });
};
    module.exports = errorHandler;
```

Use it in server.js

```
const errorHandler = require('./middleware/errorMiddleware');
app.use(errorHandler);
```

Now, anywhere in your routes, you can just throw an error, and Express will automatically pass it to this middleware.

Error Handling in Auth Routes (Example)

```
app.post('/login', async (req, res, next) => {
  const { email, password } = req.body;
  if (!email || !password)
    return next(ErrorResponse('Please enter email and password', 400));
  const user = await
  User.findOne({ email });
  if (!user) return
  next(ErrorResponse('Invalid credentials', 401));
  res.json({ message: 'Login successful' });
});
```

Final Error Flow Diagram

Client Request
↓
Express Route
↓
Route Handler (async/sync) → If error → next(err)

Response to client (JSON error)
↑
Error Middleware (Centralized Handler)
↑

Environment Variables (.env)

Don't hardcode credentials — store them in .env.

• env

```
PORT = 5000
MONGO_URI = mongodb+srv://user:
pass@cluster.mongodb.net/db
```

server.js

```
require('dotenv').config();
const PORT = process.env.PORT || 5000;
```

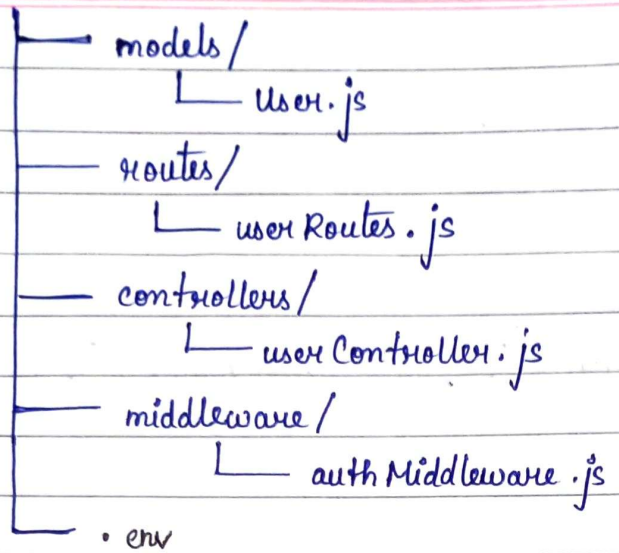
CORS Setup (for React Frontend)

When React (frontend) calls Express (backend), CORS (Cross-Origin Resource Sharing) must be allowed.

```
const cors = required('cors');
app.use(cors({
        origin: 'http://localhost:3000', // frontend URL
        credentials: true
}));
```

Express Folder Structure (Best Practice)

```
backend/
├── server.js
├── config/
│   └── db.js
```



Example

routes/userRoutes.js

```
const express = require('express');
const router = express.Router();
const { getUsers, addUser } =
require('../controllers/userController');
router.get('/', getUsers);
router.post('/', addUser);
module.exports = router;
```

controllers/userController.js

```
const User = require('../models/User');
exports.getUsers = async (req, res) => {
        const users = await User.find();
        res.json(users); };
exports.addUser = async (req, res) => {
        const newUser = new User(req.body);
```

```
    await newUser.save();
    res.status(201).json(newUser);
};
```

server.js

```
app.use('/api/users', require('./routes/userRoutes'));
```

Express with React (Full MERN)

When deployed:

- React runs on frontend (Vite or CRA)
- Express services API routes
- MongoDB stores data

In production, Express can also serve React build:

```
app.use(express.static('client/build'));
app.get('*', (req, res) => {
    res.sendFile(path.resolve(__dirname,
'client', 'build', 'index.html'));  });
```

Express with Middleware Stack (Example)

```
app.use(express.json());
app.use(cors());
app.use('/api/users', userRoutes);
app.use('/api/auth', authRoutes);
app.use(errorHandler);
```

Each route → modular, maintainable, clean code

Common Express Methods

Method	Description
app.use()	mount middleware or routes
app.get()	Handle GET request
app.post()	Handle POST request
app.put()	Handle PUT request
app.delete()	Handle DELETE request
res.json()	Send JSON response
res.status()	Set HTTP status code
next()	pass to next middleware
req.params	route params
req.query	query string
req.body	POST body.

Authentication (Brief)

Used in MERN for user login/register

Common flow :

1. User sends credentials → /login
2. Server verifies credentials
3. Server returns JWT token
4. Client stores token (local Storage)
5. Future requests include token in headers.
6. Middleware verifies token → allows access