1. Introduction to Node.js

What is Node.js?
Node.js is a **runtime environment** that allows JavaScript to run outside the browser. It's built on **Chrome's V8 engine**.

- **Non-blocking I/O** and **event-driven architecture** make it great for scalable applications.
- Uses a single-threaded **event loop** internally.
- Ideal for real-time apps like chat, APIs, dashboards.

Key Features:

- Fast execution (V8 engine)
- Asynchronous and event-driven
- Cross-platform
- Large ecosystem (npm)

Example: Simple Node Script

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

---

2. Node.js Architecture

How Node.js Works:

- Uses a **Single Threaded Event Loop** model to handle multiple clients.
- Delegates tasks like file access or DB queries to background threads.

Components:

- **V8 Engine**: Converts JS to machine code
- **Event Loop**: Handles async code
- **libuv**: C++ library for thread pool, async I/O
- **Callback Queue**: Functions waiting to run

Diagram Flow:

1. Call Stack
2. Web APIs (e.g., setTimeout)
3. Callback Queue
4. Event Loop → pushes callbacks to Call Stack

---

3. Modules in Node.js

Types:

- **Core Modules**: Built-in (e.g., `fs`, `http`)
- **Local Modules**: Your own files
- **Third-party Modules**: Installed via `npm`

Import Syntax:

```
// CommonJS (default in Node.js)
const fs = require('fs');

// ESM (need "type": "module" in package.json)
import fs from 'fs';
```

Creating Your Own Module:

```
// math.js
function add(a, b) {
  return a + b;
}
module.exports = { add };

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

---

4. npm & package.json

What is npm?
The **Node Package Manager** is used to install and manage third-party packages.

Basic Commands:

- `npm init -y` → create `package.json`
- `npm install express` → install package
- `npm uninstall express` → remove package

package.json structure:

```
{
  "name": "my-app",
```

```
  "version": "1.0.0",
  "scripts": {
    "start": "node index.js"
  },
  "dependencies": {
    "express": "^4.18.2"
  }
}
```

Local vs Global Install:

- Local: Available only in project (`node_modules`)
- Global: Available system-wide (`npm install -g`)

---

5. File System (fs Module)

Read and Write Files:

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

// Write
fs.writeFileSync('notes.txt', 'Hello Node.js');

// Read
const data = fs.readFileSync('notes.txt', 'utf-8');
console.log(data);
```

Async Version:

```
fs.writeFile('file.txt', 'Async Write', (err) => {
  if (err) throw err;
  console.log('File written!');
});

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

Other Functions:

- `fs.appendFile()`
- `fs.rename()`
- `fs.unlink()` → Delete file
- `fs.existsSync()` → Check existence

6. HTTP Module & Creating a Server

The `http` module allows you to create a basic web server without external libraries like Express.

Creating a Basic Server:

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

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

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

Key Concepts:

- `req` (IncomingMessage): holds request details (method, headers, URL).
- `res` (ServerResponse): used to send data back to the client.
- `writeHead(statusCode, headers)`: set response metadata.

---

7. EventEmitter & Streams

🔹 **EventEmitter**

Node.js uses events to handle async operations.

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

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

```
emitter.emit('greet', 'Nimit');
```

- `on()` → listen to event
- `emit()` → trigger event

🔹 **Streams**

Used to handle data chunks (useful for files, video/audio).

```
const fs = require('fs');
const readStream = fs.createReadStream('input.txt', 'utf-8');
readStream.on('data', chunk => console.log(chunk));
```

Types: Readable, Writable, Duplex, Transform

---

8. Express.js Basics

Why Express?
Express simplifies building robust web apps and APIs with routing, middleware, and request handling.

🔹 **Installing Express**

```
npm install express
```

🔹 **Creating a Basic Server**

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

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

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

Core Concepts:

- **Routing**: `.get()`, `.post()`, `.put()`, `.delete()`
- **Middleware**: Functions that run between request and response
- **Body Parsing**: `express.json()`, `express.urlencoded()`

---

9. CommonJS vs ES Modules

Feature	CommonJS (`require`)	ES Modules (`import`)
File Extension	`.js`	`.mjs` or `"type": "module"` in `package.json`
Sync vs Async	Synchronous	Asynchronous
Usage in Node.js	Default	Needs config

🔹 **Example (CommonJS)**

```
// math.js
module.exports = { add: (a, b) => a + b };

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

🔹 **Example (ESM)**

```
// math.mjs
export const add = (a, b) => a + b;

// app.mjs
import { add } from './math.mjs';
console.log(add(2, 3));
```

---

10. Promised fs & Async/Await in Server Code

Using the modern `fs/promises` module:

```
const fs = require('fs/promises');
async function readFileAsync() {
  try {
    const data = await fs.readFile('notes.txt', 'utf-8');
    console.log(data);
  } catch (err) {
    console.error(err);
  }
}
```

```
readFileAsync();
```

Using async/await in Express route:

```
app.get('/data', async (req, res) => {
  try {
    const content = await fs.readFile('data.json', 'utf-8');
    res.json(JSON.parse(content));
  } catch (err) {
    res.status(500).send('Error reading file');
  }
});
```

Why use Promises & async/await?

- Cleaner syntax than callbacks
- Easier to handle errors with `try/catch`
- Avoid callback hell

11. Testing Fundamentals

🔹 **Why Test?**

- Ensures code correctness and prevents regressions
- Helps during refactoring and team collaboration

🔹 **Types of Testing**

- **Unit Tests**: Test individual functions
- **Integration Tests**: Test multiple units/modules together
- **End-to-End (E2E)**: Test full workflows

🔹 **Jest Basics**

Installation

```
npm install --save-dev jest
```

Sample Test (math.js)

```
function add(a, b) {
  return a + b;
}
module.exports = { add };
```

Test File (math.test.js)

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

test('adds two numbers', () => {
  expect(add(2, 3)).toBe(5);
});
```

Run tests

```
npx jest
```

🔹 **Mocking API Calls**

Use `jest.mock()` to mock functions/modules:

```
jest.mock('axios');
axios.get.mockResolvedValue({ data: { message: "Mocked" } });
```

---

12. Security Basics in JS Apps

🔹 **1. XSS (Cross-site Scripting)**

What: Attacker injects malicious JS in user inputs

Prevention:

- Escape HTML before rendering
- Use libraries like `DOMPurify`, or server-side validation

🔹 **2. CSRF (Cross-site Request Forgery)**

What: Malicious site triggers unintended actions using your logged-in session

Prevention:

- Use anti-CSRF tokens
- SameSite cookies
- Check `Referer`/`Origin` headers

🔹 **3. Token Handling (JWT)**

Best Practices:

- Store **access tokens** in HttpOnly cookies (safer than localStorage)
- Use **short expiry** access tokens and refresh tokens
- Always validate tokens on the server

🔹 **4. Password Hashing**

```
const bcrypt = require('bcrypt');
const hashed = await bcrypt.hash('mypassword', 10);
const valid = await bcrypt.compare('mypassword', hashed);
```

---

13. Environment Variables & dotenv

🔹 **Why Use `.env`?**

To keep secrets/configs (API keys, DB passwords) outside the codebase.

```
PORT=3000
DB_PASSWORD=supersecret
```

Usage in Code:

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

Install dotenv

```
npm install dotenv
```

Always add `.env` to `.gitignore`.

---

14. Deployment Fundamentals

🔹 **Steps to Deploy a Node App (e.g., to Render or Railway)**

1. Push to GitHub
2. Connect repo to platform (Render, Vercel, etc.)
3. Set build/start commands
4. Set environment variables

Start Script in package.json

```
"scripts": {
  "start": "node index.js"
}
```

🔹 **Using PM2 for Production**

```
npm install -g pm2
pm2 start index.js
```

Benefits:

- Keeps app alive (auto-restart)
- Logs management
- Cluster mode for scaling

---

15. MVC Pattern in Node.js

MVC = Model - View - Controller

🔹 **Why use MVC?**

- Clean separation of concerns
- Scales better for large apps

🔹 **Structure Example**

```
/models/User.js
/controllers/userController.js
/routes/userRoutes.js
/server.js
```

Sample:

```
models/User.js

const mongoose = require('mongoose');

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

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

controllers/userController.js

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

exports.getUsers = async (req, res) => {
  const users = await User.find();
```

```
  res.json(users);
};
```

routes/userRoutes.js

```
const express = require('express');
const router = express.Router();
const { getUsers } = require('../controllers/userController');

router.get('/users', getUsers);

module.exports = router;
```

server.js

```
const express = require('express');
const app = express();
const userRoutes = require('./routes/userRoutes');

app.use(express.json());
app.use('/api', userRoutes);
app.listen(3000);
```

16. Connecting Node.js to MongoDB

🔹 **Using Mongoose (ODM for MongoDB)**

Installation

```
npm install mongoose
```

Connecting

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

🔹 **Defining a Schema & Model**

```
const userSchema = new mongoose.Schema({
  name: String,
```

```
  email: { type: String, unique: true },
  createdAt: { type: Date, default: Date.now }
});

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

🔹 **CRUD Example**

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

// Read
const users = await User.find();

// Update
await User.updateOne({ email: "john@example.com" }, { name: "Johnny"
});

// Delete
await User.deleteOne({ email: "john@example.com" });
```

---

17. File Uploads & Streams

🔹 **File Upload with `multer`**

Install

```
npm install multer
```

Usage

```
const multer = require('multer');
const upload = multer({ dest: 'uploads/' });

app.post('/upload', upload.single('file'), (req, res) => {
  res.send(req.file);
});
```

🔹 **Streaming Files with `fs`**

```
const fs = require('fs');
const readStream = fs.createReadStream('large.txt');
```

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

Pipe to Response (Efficient File Download)

```
app.get('/download', (req, res) => {
  const stream = fs.createReadStream('bigfile.pdf');
  stream.pipe(res);
});
```

---

18. Sending Emails in Node.js

- **Using Nodemailer**

Install

```
npm install nodemailer
```

Setup

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

const transporter = nodemailer.createTransport({
  service: 'gmail',
  auth: {
    user: 'your@email.com',
    pass: 'your-app-password'
  }
});

const mailOptions = {
  from: 'you@email.com',
  to: 'target@email.com',
  subject: 'Test Email',
  text: 'Hello from Node.js!'
};

transporter.sendMail(mailOptions)
  .then(() => console.log("Email sent!"))
```

```
  .catch(console.error);
```

---

19. Custom Middleware in Express

🔹 **What is Middleware?**

A function that runs between the request and the final route handler.

🔹 **Creating Custom Middleware**

```
const logger = (req, res, next) => {
  console.log(`${req.method} ${req.path}`);
  next();
};

app.use(logger);
```

🔹 **Types**

- **Application-level**: `app.use(logger)`
- **Route-level**: `app.get('/route', middleware, handler)`
- **Error-handling**: `(err, req, res, next) => { ... }`

🔹 **Built-in Middleware**

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

---

20. Authentication with JWT (JSON Web Token)

🔹 **Install**

```
npm install jsonwebtoken bcrypt
```

🔹 **Create JWT**

```
const jwt = require('jsonwebtoken');
const token = jwt.sign({ userId: user._id }, process.env.JWT_SECRET,
{
  expiresIn: '1h'
});
```

🔹 **Middleware to Protect Routes**

```
const authMiddleware = (req, res, next) => {
  const token = req.headers.authorization?.split(" ")[1];
  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded;
    next();
  } catch {
    return res.status(401).json({ error: 'Unauthorized' });
  }
};

app.get('/protected', authMiddleware, (req, res) => {
  res.send("You are in!");
});
```

🔹 **Password Hashing with `bcrypt`**

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

// Hash
const hashedPassword = await bcrypt.hash('mypassword', 10);

// Compare
const isValid = await bcrypt.compare('mypassword', hashedPassword);
```

21. Deploying Node.js Apps (Local + Cloud)

🔹 **Common Deployment Targets**

- **Local**: using PM2 or `node index.js`
- **Cloud**:
  - Render, Railway, Cyclic (easy)
  - EC2/VPS (manual)
  - Vercel (for serverless APIs)
  - Docker containers

🔹 **Basic Render Deployment Steps**

1. Push your code to GitHub
2. Go to render.com, connect GitHub repo
3. Add environment variables in the dashboard
4. Select Node.js and deploy

🔹 **Production Tips**

- Use `.env` for secrets
- Avoid exposing logs
- Set proper `PORT`
- Use a reverse proxy like **nginx** in custom VPS setup

---

22. Environment Setup & .env

🔹 **Using `dotenv`**

```
npm install dotenv
```

In `index.js`:

```
require('dotenv').config();

const port = process.env.PORT || 3000;
```

In `.env`:

```
PORT=5000
MONGO_URI=your-mongo-uri
JWT_SECRET=your-secret
```

> 🔐 Never push `.env` to GitHub — add it to `.gitignore`.

---

23. Logging in Node.js

🔹 **Using `console` (Basic)**

```
console.log('App started');
console.error('Something went wrong');
```

🔹 **Using `winston` (Production Logger)**

```
npm install winston
```

js
CopyEdit

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

const logger = winston.createLogger({
```

```
  transports: [
    new winston.transports.File({ filename: 'logs/error.log', level:
'error' }),
    new winston.transports.Console()
  ]
});

logger.info('Server running...');
logger.error('Error occurred');
```

---

24. Error Handling Patterns

🔹 **Try-Catch (Async/Await)**

```
app.get('/route', async (req, res) => {
  try {
    const data = await somethingAsync();
    res.json(data);
  } catch (err) {
    res.status(500).json({ error: 'Something went wrong' });
  }
});
```

🔹 **Centralized Error Middleware**

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send("Internal Server Error");
});
```

🔹 **Throwing Errors**

```
if (!user) throw new Error("User not found");
```

---

25. WebSockets with Socket.io

🔹 **Install and Setup**

```
npm install socket.io
```

Server:

```
const http = require('http');
const { Server } = require('socket.io');

const server = http.createServer(app);
const io = new Server(server);

io.on('connection', socket => {
  console.log('User connected:', socket.id);

  socket.on('chat', msg => {
    io.emit('chat', msg); // broadcast
  });

  socket.on('disconnect', () => {
    console.log('User disconnected');
  });
});

server.listen(3000);
```

Client:

```
<script src="/socket.io/socket.io.js"></script>
<script>
  const socket = io();

  socket.on('chat', msg => {
    console.log('Received:', msg);
  });

  socket.emit('chat', 'Hello World!');
</script>
```

🔹 **Use Cases**

- Real-time chat
- Live notifications
- Multiplayer games
- Collaborative apps (e.g., whiteboard)