upGrad ENTERPRISE

Node.js Fundamentals – Day 1

Siddhesh Prabhugaonkar
Microsoft Certified Trainer
siddheshpg@azureauthority.in

Agenda

1. Introduction to Node.js and its architecture
2. Event loop & asynchronous programming
3. Core modules (fs, path, http, etc.)
4. npm & package.json basics
5. Writing and running simple Node.js scripts

Pre-requisite

- VS Code
- Postman
- Oracle
- Docker desktop
- AWS account
- git for Windows

Getting Started with Node.js (Setup & Environment)

Objective: Set up Node.js environment and verify installation.

Step-by-Step Setup:

1. **Install Node.js**
   - Go to https://nodejs.org
   - Download the **LTS version** (recommended for stability)
   - Follow the installer prompts (keep default settings)
2. **Verify Installation**

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

   - Output should show Node.js and npm versions (e.g., `v20.11.1`, `9.8.0`)
5. **Set up your IDE**
   - Recommended: **Visual Studio Code**
   - Extensions:
     - *ESLint* (code quality)
     - *Code Runner* (run JS directly)
     - *Path Intellisense*

- *Prettier* (auto-format)

6. **First Program**

```
mkdir node-fundamentals
cd node-fundamentals
```

Create a file `app.js`:

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

Run:

```
node app.js
```

Output:

```
Hello from Node.js!
```

Tip: Unlike browsers, Node.js executes JavaScript directly on your machine, outside a web page.

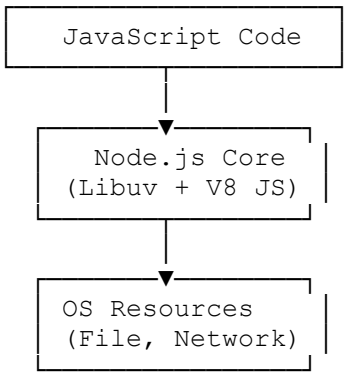
Introduction to Node.js and its Architecture

- Node.js is a **runtime environment** for executing JavaScript outside the browser.
- Built on **Google's V8 engine** (the same engine Chrome uses).
- Ideal for **server-side programming**, especially for **I/O-heavy applications** (e.g., APIs, chat servers).

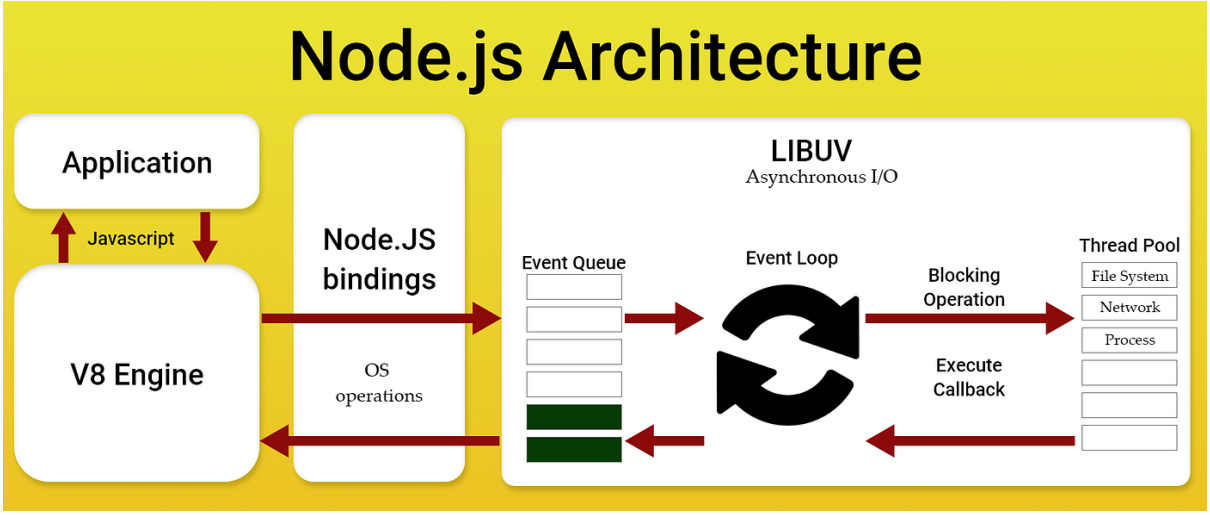
Key Features:

- Single-threaded, non-blocking architecture
- Event-driven model
- Asynchronous execution
- Cross-platform (Windows, macOS, Linux)

Node.js Architecture:



```
          +----------------+
          |   Application  |
          +--------+-------+
                   |
           Event Loop (libuv)
                   |
     +-------------+-------------+
     |             |             |
Non-blocking   Thread Pool   System APIs
     |             |             |
  Callbacks   I/O Tasks   Network Calls
```



Why Node.js?

- Great for real-time apps (chat, games, streaming)
- Uses same language (JavaScript) for client + server
- Huge ecosystem via npm

Activity:
Discuss examples of applications where Node.js would be more efficient than traditional multi-threaded backends (like Java or PHP).

- Chat apps, online gaming servers, collaborative editing tools (like Google Docs), or live sports tickers.
- API Gateways and Microservices
- Video or audio streaming services, real-time data processing pipelines, or large file uploads.

Choose Node.js when your application's main bottleneck is waiting for I/O (network, database), not when it's doing heavy math or complex calculations (CPU-bound).

Can do with Node.js

- Node.js can generate dynamic page content.
- Node.js can create, open, read, write, delete, and close files on the server.
- Node.js can collect form data.
- Node.js can perform CRUD operations on your database.

CANNOT do with Node.js

- Node is a platform for writing JavaScript applications outside web browsers.
- There is no DOM built into Node, nor any other browser capability.
- Node can't run on GUI, but run on terminal
    - Offers basic REPL capability

Poll Question

Q: Node.js runs JavaScript in the browser.

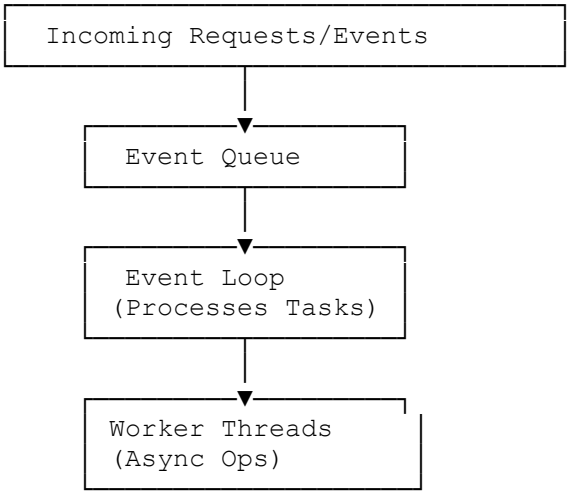
True / False

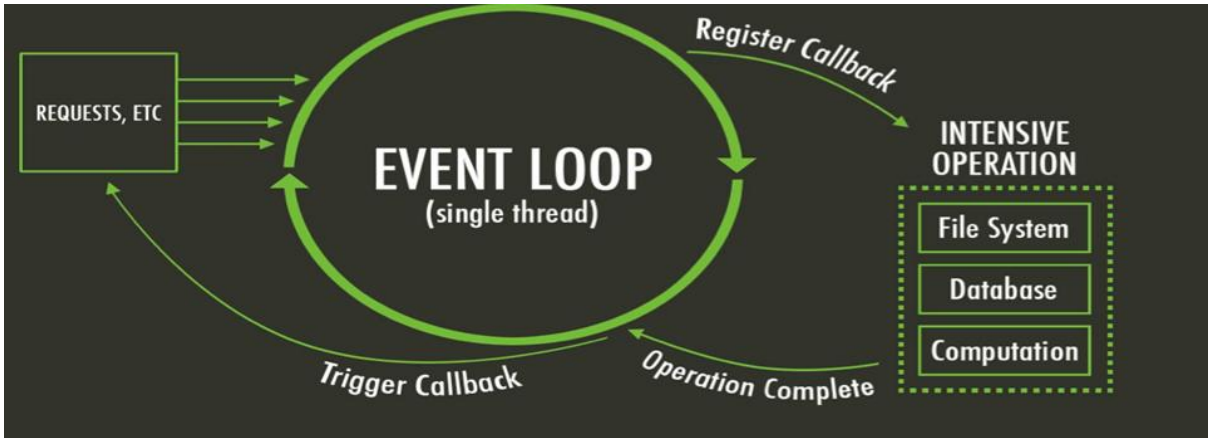
Event Loop & Asynchronous Programming

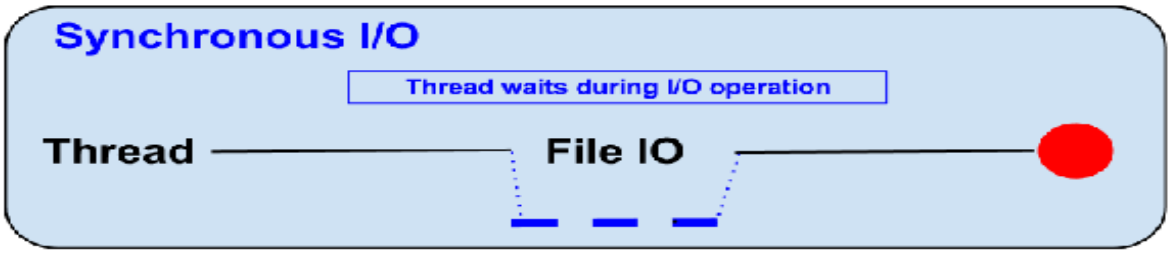
- Node.js uses an **event-driven, non-blocking I/O model**.
- The **event loop** handles multiple requests efficiently on a single thread.

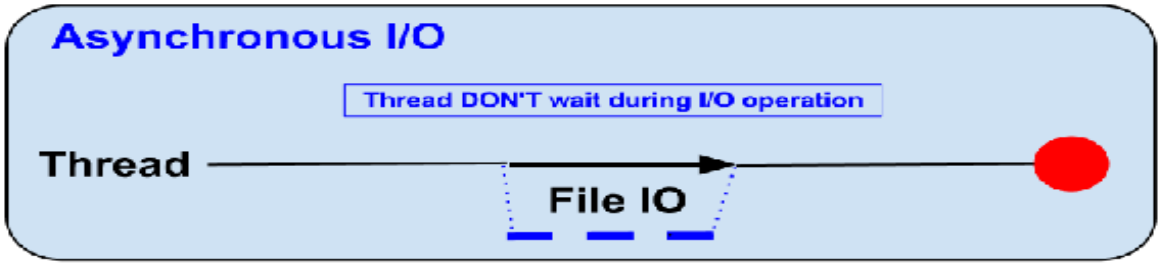
Analogy:
Think of a restaurant with one waiter who takes multiple orders but doesn't wait for food to be cooked before serving others.

Event Loop Flow:









Code Example (Synchronous vs Asynchronous):

```
// Synchronous
console.log("Start");
for (let i = 0; i < 3; i++) {
  console.log(i);
}
console.log("End");

// Asynchronous
console.log("Start");
setTimeout(() => console.log("Async Task Done"), 1000);
console.log("End");
```

Output:

```
Start
End
Async Task Done
```

Tip:

- Use callbacks, promises, or async/await for asynchronous code.

- Avoid blocking operations (like `readFileSync`).

Practice Exercise:
Write a script that reads a file asynchronously and logs its content.

```
const fs = require("fs");
fs.readFile("test.txt", "utf8", (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

This will give error : **ERR_REQUIRE_ESM**

This is a common issue! The error `ERR_REQUIRE_ESM` means your project (which uses CommonJS's `require()`) is trying to import a package (`chalk` version 5) that is **only** available as an ES Module (ESM).

There are 2 ways to fix this

- **CommonJS (CJS):** This is the "classic" Node.js way of working. It uses `require()` and `module.exports`. Your current `index.js` file is a CommonJS module.
- **ES Modules (ESM):** This is the **official, modern standard** for JavaScript, used by browsers and now fully supported in Node.js. It uses `import` and `export`.

The package `chalk` (starting with version 5) was rewritten to *only* use ESM.

ESM is recommended because it's the universal standard for JavaScript. The entire ecosystem is moving to it, and it offers advantages like static analysis, better optimization, and compatibility with browser-based code.

Option 1: Keep CommonJS and Use Dynamic Import()

This solution keeps your project as CommonJS but uses a special `import()` function (which is asynchronous) to load the ESM `chalk` package.

1. Keep `package.json` as is. (No changes needed)

2. Modify `index.js`

You can't use `require("chalk")`. Instead, you must use the `import()` function, which returns a promise. The easiest way is to wrap your code in an `async` function.

```
// index.js

// Note: We can't use 'const chalk = require("chalk")'
```

```
// 1st way
(async () => {
  const chalk = (await import('chalk')).default;
  console.log(chalk.green("NPM is working!"));
  console.log(chalk.blue("All systems go!"));
  console.log(chalk.red("Error: Something went wrong!"));
})();

// 2nd way
async function main() {
  const chalk = (await import("chalk")).default;

  console.log(chalk.green("NPM is working!"));
  console.log(chalk.blue("All systems go!"));
  console.log(chalk.red("Error: Something went wrong!"));
}

main();
```

Option 2: Convert Your Project to ESM (Recommended)

1. Modify `package.json`:

Add the line `"type": "module"` to tell Node.js your code is an ES Module.

JSON

```
{
  "name": "src",
  "version": "1.0.0",
  "main": "index.js",
  "type": "module",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": "",
  "dependencies": {
    "chalk": "^5.6.2"
  }
}
```

2. Modify `index.js`:

```
// Use the ESM 'import' syntax instead of require()
import chalk from "chalk";

console.log(chalk.green("NPM is working!"));
console.log(chalk.blue("All systems go!"));
console.log(chalk.red("Error: Something went wrong!"));
```

run `node index.js`

Poll Question

Q: Why doesn't the timeout block code execution?

A: Because Node.js uses a **non-blocking event loop**

Core Modules Node.js includes several built-in modules.

Module	Purpose
`fs`	File System operations
`path`	Handle file paths
`http`	Create HTTP servers
`os`	System info

File System (fs):

```
import fs from "fs";
fs.writeFileSync("demo.txt", "Hello Node!");
console.log(fs.readFileSync("demo.txt", "utf8"));
```

Path:

```
import path, { dirname } from "path";
import { fileURLToPath } from "url";

// This is the ESM way to get the current directory name
const __filename = fileURLToPath(import.meta.url);
const __dirname = dirname(__filename);

const filePath = path.join(__dirname, "demo.txt");

console.log("File path is:", filePath);
```

HTTP Server:

```
import http from "http";
const server = http.createServer((req, res) => {
  res.end("Hello from Node.js Server!");
});
server.listen(3000, () => console.log("Server running on port 3000"));
```

Assignment:
Create a Node.js server that serves your name on `localhost:3000`.

Tip:
Always handle errors in `fs` and `http` modules using callbacks or `try...catch`.

Poll Question

Q: Which module helps create an HTTP server?

---

npm & package.json Basics

- **npm (Node Package Manager)** allows you to install and manage packages.
- **package.json** stores project metadata and dependencies.

Steps:

1. Initialize a project
   `npm init -y`
2. Install a package
   `npm install chalk`
3. Use it in your script: index.js

```
import chalk from "chalk";
console.log(chalk.green("NPM is working!"));
console.log(chalk.blue("All systems go!"));
console.log(chalk.red("Error: Something went wrong!"));
```

package.json Example:

```
{
  "name": "node-demo",
  "version": "1.0.0",
  "main": "app.js",
  "dependencies": {
    "chalk": "^5.3.0"
  }
}
```

Tip:

- Use `npm install --save-dev` for development-only dependencies.
- Run scripts via `"scripts"` property.

Activity:
Add a script `"start": "node index.js"` and run it using:

```
npm start
```

Writing and Running Simple Node.js Scripts

Example: System Info Script

```
import os from "os";

console.log("Platform:", os.platform());
console.log("Architecture:", os.arch());
console.log("CPU Cores:", os.cpus().length);
```

Example: Simple Calculator

```
const args = process.argv.slice(2);
const [a, b] = args.map(Number);
console.log(`Sum = ${a + b}`);
```

Run:

```
node calc.js 5 10
```

Example: Write a small CLI app using Node.js fundamentals.

```
import readline from "readline";

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout,
});

rl.question("What is your name? ", (name) => {
  console.log(`Hello, ${name}! Welcome to Node.js`);
  rl.close();
});
```

Run:

```
node greet.js
```

Output:

```
What is your name? Siddhesh
Hello, Siddhesh! Welcome to Node.js
```

Assignment:
Extend the script to ask for two numbers and print their sum.

Summary

- Node.js = JavaScript runtime powered by V8
- Event-driven, asynchronous model is key to scalability
- Core modules like `fs`, `path`, `http` are essential building blocks
- npm helps manage packages efficiently
- Simple scripts can automate tasks or power servers

Assignments:

- Build a simple HTTP server that reads a local HTML file and serves it.
- Explore npm packages like `axios` or `moment`.