Lecture 1: Introduction to Node.js

Introduction to Node.js

Node.js is an open-source, cross-platform runtime environment that allows developers to execute JavaScript code outside of a web browser. It is built on **Chrome's V8 JavaScript engine** and is widely used for building scalable network applications, real-time applications, and backend services. Unlike traditional JavaScript, which runs in the browser, Node.js enables developers to use JavaScript on the server side.

History of Node.js

Node.js was created by **Ryan Dahl** in **2009** to improve the way JavaScript could handle I/O operations asynchronously. Before Node.js, JavaScript was mainly used for frontend development within the browser. Dahl's goal was to create a lightweight and efficient environment where JavaScript could be used on the server.

Key Milestones in Node.js Development:

- **2009:** Ryan Dahl released the first version of Node.js.
- **2010:** npm (Node Package Manager) was introduced, making it easy to manage dependencies.
- **2014:** The io.js fork was created, introducing faster updates and improvements.
- **2015:** Node.js and io.js merged, and the Node.js Foundation was formed.
- **2018:** The OpenJS Foundation was established to govern Node.js.
- **2023-Present:** Node.js continues to evolve with new features, performance improvements, and security updates.

Features of Node.js

Node.js is widely used for building high-performance web applications and APIs due to its powerful features. Here are some key features that make Node.js stand out:

1. Asynchronous and Event-Driven

- Node.js uses a **non-blocking I/O** model, meaning it can handle multiple requests simultaneously without waiting for one to finish before starting another.
- Uses an **event-driven architecture** where callbacks and event loops manage execution efficiently.

2. Single-Threaded but Highly Scalable

- Unlike traditional multi-threaded models, Node.js operates on a **single-threaded event loop** that can handle thousands of concurrent connections efficiently.
- Uses **libuv** for handling asynchronous operations like file system access, networking, and timers.

3. Fast Execution with V8 Engine

- Node.js runs on **Google's V8 JavaScript engine**, which **compiles JavaScript to machine code** for faster execution.
- Highly optimized for performance, making it suitable for real-time applications.

4. Cross-Platform Support

- Node.js runs on **Windows, macOS, and Linux** without modification.
- Supports writing native modules in **C++** for high-performance computing needs.

5. Built-in Package Manager (npm)

- Comes with **npm (Node Package Manager)**, the world's largest ecosystem of open-source libraries.
- Developers can easily install and manage dependencies using npm.

6. Supports Microservices and APIs

- Easily build **RESTful APIs** and **microservices** for scalable architectures.
- Supports frameworks like **Express.js, NestJS, and Fastify** to simplify development.

7. Real-time Capabilities with WebSockets

- Ideal for applications requiring **real-time updates** (e.g., chat apps, live streaming, multiplayer games).
- Uses **WebSockets** and libraries like **Socket.io** to enable bidirectional communication.

8. File System and Stream Handling

- Provides **fs module** for handling files asynchronously.
- Uses **streams** for efficient data handling, making it great for large file processing.

9. High Community Support

- Large **open-source community** with constant updates and improvements.
- Actively maintained by **OpenJS Foundation**.

10. Security and Performance Enhancements

- Supports **HTTPS, JWT authentication, OAuth, and data encryption**.
- Frequent **security updates** for vulnerabilities.

Installing Node.js:

You can download the latest version from the official Node.js website.

Basic Node.js Syntax:

Create a new file called hello.js and add the following code:

```
console.log('Hello, World!');
```

Running a Node.js Program:

```
node hello.js
```

Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. It allows developers to run JavaScript on the server-side, making it a popular choice for web development.

Node.js Ecosystem

The **Node.js ecosystem** is a vast and thriving environment of tools, frameworks, and libraries that help developers build powerful and scalable applications. It includes **runtime, package management, frameworks, libraries, databases, deployment tools, and community support**.

1. Node.js Runtime

- The core of the ecosystem, built on **Google's V8 JavaScript engine**.
- Provides **APIs** for handling file systems, networking, and other system functionalities.
- Supports **asynchronous programming** using event-driven architecture.

2. npm (Node Package Manager)

- The **largest package manager** in the world with over **2 million+ open-source packages**.
- Helps in managing dependencies, installing third-party libraries, and running scripts.
- Commands:
  - Install a package: `npm install <package-name>`
  - Initialize a project: `npm init`
  - Install dependencies: `npm install`

Example: Install Express.js

```
npm install express
```

3. Popular Node.js Frameworks & Libraries

Backend Frameworks

- **Express.js** – Minimal and flexible framework for building APIs.
- **NestJS** – A modular and scalable framework for enterprise applications.
- **Fastify** – A fast and lightweight web framework.
- **Koa.js** – A modern alternative to Express with middleware-based architecture.

Real-Time Communication

- **Socket.io** – Enables real-time, bidirectional communication (chat apps, notifications).
- **WebSockets** – Standard protocol for real-time web applications.

Database Libraries

- **Mongoose** – ODM for MongoDB.
- **Sequelize** – ORM for SQL databases.
- **TypeORM** – TypeScript-friendly ORM for SQL databases.
- **Knex.js** – SQL query builder.

Testing Frameworks

- **Jest** – For unit and integration testing.
- **Mocha & Chai** – Flexible testing framework.
- **Supertest** – API testing.

4. Frontend Integration

Node.js works seamlessly with frontend technologies:

- **Angular, React, Vue.js** – Used for frontend development.
- **Next.js, Nuxt.js** – Server-side rendering frameworks for React and Vue.
- **Webpack, Vite** – Build tools for frontend optimization.

5. Package and Task Runners

- **npm scripts** – Automate tasks (e.g., `npm run start`).
- **Yarn** – Alternative to npm with better performance.
- **PM2** – Process manager for running Node.js applications.

6. Databases Compatible with Node.js

- **NoSQL Databases**: MongoDB, CouchDB, Firebase.
- **SQL Databases**: MySQL, PostgreSQL, MariaDB.
- **Graph Databases**: Neo4j.

Example: Connecting Node.js with MongoDB

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

mongoose.connect('mongodb://localhost:27017/mydb', { useNewUrlParser: true,
useUnifiedTopology: true })
    .then(() => console.log('Database connected'))
    .catch(err => console.error(err));
```

7. DevOps, Deployment & Cloud Services

- **Docker & Kubernetes** – Containerizing Node.js apps.
- **AWS, Azure, Google Cloud** – Hosting and cloud computing.
- **Heroku, Vercel, Netlify** – Easy deployment platforms.
- **CI/CD Tools**: Jenkins, GitHub Actions, Travis CI.

8. Security and Authentication

- **Helmet.js** – Secures HTTP headers.
- **bcrypt.js** – Password hashing.
- **JWT (jsonwebtoken)** – Authentication with JSON Web Tokens.
- **OAuth2 & Passport.js** – Authentication strategies.

Example: Using JWT for authentication

```
const jwt = require('jsonwebtoken');

const token = jwt.sign({ userId: 123 }, 'secretKey', { expiresIn: '1h' });
console.log(token);
```

9. Monitoring & Performance Tools

- **New Relic, AppDynamics** – Application monitoring.
- **Winston, Bunyan** – Logging libraries.
- **Node.js Inspector** – Debugging tool.

10. Community and Support

- **Node.js Foundation & OpenJS Foundation** – Maintains Node.js development.
- **GitHub, Stack Overflow, Dev.to** – Community discussions and resources.
- **Meetups, Conferences, Hackathons** – Events like Node.js Summit, JSConf.

1. Introduction to Node.js Modules

A **module** in Node.js is a reusable block of code that encapsulates related functionality. It helps in organizing code into smaller, manageable pieces.

Types of Modules in Node.js

1. **Core Modules** – Built-in modules like `fs`, `http`, `path`.
2. **Local Modules** – Custom user-defined modules.
3. **Third-Party Modules** – Installed via npm (e.g., Express, Mongoose).

Example of Core Module (fs - File System)

```
const fs = require('fs');
fs.writeFileSync('hello.txt', 'Hello, Node.js!');
console.log('File created successfully');
```

2. Creating and Using Node.js Modules

Node.js allows you to create your own modules using `module.exports` and `require()`.

Creating a Custom Module (math.js)

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

function multiply(a, b) {
    return a * b;
}

module.exports = { add, multiply };
```

Using the Custom Module (app.js)

```
const math = require('./math');

console.log(math.add(5, 3)); // Output: 8
console.log(math.multiply(4, 2)); // Output: 8
```

3. Introduction to Node.js Packages

A **package** in Node.js is a collection of files that provides functionality to an application.

- Packages can be installed using `npm` (Node Package Manager).
- A package usually contains a **package.json** file that describes the package.

Example of a package: `express` (for web servers)

4. Creating and Using Node.js Packages

Creating a New Package

1. Initialize a new package:

```
npm init -y
```

This creates a `package.json` file.

2. Create a main file (`index.js`) and write the package logic.

Example: Creating a simple logging package (`logger.js`)

```
function logMessage(msg) {
    console.log(`[LOG]: ${msg}`);
}

module.exports = logMessage;
```

Using the package in another file:

```
const log = require('./logger');
log('Hello from my custom package!');
```

5. Understanding the package.json File

The `package.json` file is the heart of a Node.js project. It contains:

- **name** – Package name
- **version** – Current version
- **description** – Description of the package
- **main** – Entry file
- **dependencies** – Installed packages
- **scripts** – Custom scripts

Example package.json File:

```
{
  "name": "my-app",
  "version": "1.0.0",
  "description": "A sample Node.js app",
  "main": "index.js",
  "scripts": {
    "start": "node index.js"
  },
  "dependencies": {
    "express": "^4.18.2"
  }
}
```

6. Using npm to Manage Dependencies

Installing Packages

- Install a package:

```
npm install express
```

- Install as a dev dependency:

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

- Install all dependencies from `package.json`:

  ```
  npm install
  ```

Removing Packages

```
npm uninstall express
```

Updating Packages

```
npm update
```

Checking Installed Packages

```
npm list
```

7. Introduction to Popular Node.js Packages

Package Name	Description
Express.js	Fast and minimal web framework
Mongoose	MongoDB ORM for handling database queries
Socket.io	Real-time bidirectional communication
dotenv	Loads environment variables from a `.env` file
cors	Middleware for handling CORS
jsonwebtoken	Used for authentication with JWT
bcryptjs	Hash passwords securely
nodemailer	Send emails in Node.js
moment.js	Work with dates and time
axios	HTTP request library

Example: Using Express.js for a Simple Server

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

app.get('/', (req, res) => {
    res.send('Hello, World!');
```

```
});

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

Event Loop

The **Event Loop** is a fundamental concept in JavaScript (and Node.js) that enables **asynchronous, non-blocking** operations, making JavaScript efficient for handling I/O-heavy tasks like **network requests and file operations**.

How the Event Loop Works

JavaScript uses a **single-threaded** execution model, meaning it processes one task at a time. However, it can handle asynchronous operations using the **Event Loop**, which manages the execution of multiple operations in an orderly manner.

Key Components of the Event Loop

1. **Call Stack** – Executes synchronous code (function calls, loops, etc.).
2. **Web APIs** – Handles asynchronous operations like `setTimeout()`, HTTP requests, and event listeners.
3. **Callback Queue** – Stores callbacks from asynchronous tasks, waiting to be executed.
4. **Microtask Queue** – Contains higher-priority tasks like **Promises and** **`process.nextTick()`** in Node.js.
5. **Event Loop** – The mechanism that moves tasks from the callback queue to the call stack when it is empty.

Event Loop Execution Process

1. The **Call Stack** runs synchronous code first.
2. If it encounters an **asynchronous operation**, it delegates it to the **Web APIs** (e.g., `setTimeout`, `fetch`).
3. Once the asynchronous task is complete, the callback function is pushed to the **Callback Queue**.
4. **Microtasks (Promises, `process.nextTick()`)** are prioritized over callback queue tasks.
5. The **Event Loop** continuously checks if the Call Stack is empty and then pushes tasks from the **Microtask Queue** first, followed by the **Callback Queue**.

Example: Understanding the Event Loop

```
console.log("Start"); // 1. Executes immediately

setTimeout(() => {
    console.log("Timeout Callback"); // 4. Executed after synchronous code
and microtasks
}, 0);

Promise.resolve().then(() => {
    console.log("Promise Resolved"); // 3. Executes before setTimeout due
to microtask queue
});

console.log("End"); // 2. Executes immediately
```

Output:

```
Start
End
Promise Resolved
Timeout Callback
```

Explanation:

1. `"Start"` is printed first (synchronous code).
2. `"End"` is printed next (synchronous code).
3. The **Promise** resolves before the `setTimeout()` callback because promises go into the **Microtask Queue**, which has higher priority than the **Callback Queue**.
4. Finally, the `setTimeout()` callback is executed.

Where is the Event Loop Used?

Browsers – Handles user interactions, animations, network requests.
Node.js – Manages asynchronous file I/O, database queries, and server requests.

Summary

- JavaScript executes **synchronous code first**.
- Asynchronous operations (e.g., `setTimeout`, `fetch`) go through **Web APIs**.
- **Promises & process.nextTick()** have **higher priority** than callbacks.
- The **Event Loop** moves tasks from the callback queue to the call stack when it's empty.

Let's consider a **real-world example** of the Event Loop in **Node.js** using a **file read operation**.

Scenario: Processing a Large File Asynchronously

Imagine you are building a web application where users can **upload large files**, and you need to **read the file content without blocking** other operations.

Node.js provides non-blocking file operations using `fs.readFile()`.

Let's see how the **Event Loop** handles this:

Example: Reading a File and Processing Other Tasks

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

console.log("1. Start reading the file...");

// Asynchronous File Read
fs.readFile("example.txt", "utf-8", (err, data) => {
    if (err) throw err;
    console.log("3. File content read successfully!");
});

// Promise to simulate a database query
Promise.resolve().then(() => console.log("2. Database query completed"));

// Simulate an HTTP request handling
setTimeout(() => {
    console.log("4. HTTP request processed");
}, 0);

console.log("5. End of script execution.");
```

Expected Output

```
1. Start reading the file...
5. End of script execution.
2. Database query completed
3. File content read successfully!
4. HTTP request processed
```

How the Event Loop Works in this Example

1. **Synchronous Code Executes First:**
    - `"1. Start reading the file..."` is printed.
    - `fs.readFile()` is called, but since it's **asynchronous**, it is handled by **Node.js I/O** and won't block execution.
    - `"5. End of script execution."` is printed because it's synchronous.
2. **Microtask Queue (Promises) Executes Next:**
    - `"2. Database query completed"` is printed because Promises have a higher priority in the Event Loop.

3. **I/O Callback Executes After the Microtask Queue:**
    - Once `fs.readFile()` finishes reading the file, the callback function is pushed to the Callback Queue.
    - `"3. File content read successfully!"` is printed.
4. **setTimeout() Executes Last:**
    - The `setTimeout()` callback is executed last, printing `"4. HTTP request processed"`, since it belongs to the **Callback Queue** and runs only after microtasks and I/O callbacks.

Real-World Application of the Event Loop

Non-blocking file operations: Handling file uploads without blocking other user requests.
Asynchronous database queries: Executing multiple database operations in parallel.
Handling HTTP requests: Ensuring the server remains responsive while processing multiple API calls.

Stack, Heap, and Queue in Memory Management

In programming, **Stack, Heap, and Queue** are different ways of organizing and managing memory. Let's break them down with **real-world analogies** and **examples**.

1 (LIFO – Last In, First Out)

Definition:
The stack is a **fixed-size** memory structure used for storing **function calls, local variables, and execution context** in a LIFO (Last In, First Out) order.

Real-World Analogy:
Think of a **stack of plates** in a cafeteria. You can only add or remove plates from the **top**.

Characteristics:

- Stores **local variables**, function calls, and return addresses.
- Grows and shrinks **automatically** as functions are called and returned.
- **Fast access**, but **limited memory**.
- **Data is removed automatically** when a function completes.

Example in JavaScript (Stack Memory)

```
function firstFunction() {
    secondFunction();
}

function secondFunction() {
    console.log("Inside second function");
}
```

```
firstFunction();
console.log("Execution finished");
```

Stack Execution Flow:

1. `firstFunction()` is pushed onto the stack.
2. `secondFunction()` is called inside `firstFunction()`, so it is pushed onto the stack.
3. `console.log("Inside second function")` executes and `secondFunction()` is popped from the stack.
4. `firstFunction()` finishes and is popped from the stack.
5. `console.log("Execution finished")` executes.

Output:

```
Inside second function
Execution finished
```

2 Heap (Dynamic Memory Allocation)

Definition:
The Heap is a **large, unstructured memory area** used for storing **objects and dynamically allocated variables**.

Real-World Analogy:
Think of a **warehouse** where you can store items anywhere and retrieve them with a reference.

Characteristics:

- Used for **dynamic memory allocation** (objects, arrays).
- Variables **stay in memory until garbage collected**.
- **Slower** compared to the stack, but offers **more flexibility**.

Example in JavaScript (Heap Memory)

```
let person1 = { name: "Alice", age: 25 };
let person2 = person1;

person2.age = 30;
console.log(person1.age); // Output: 30
```

Heap Memory Behavior:

- `person1` is an **object** stored in the heap.
- `person2` **references** the same memory location.
- Modifying `person2.age` also changes `person1.age`, since both point to the same memory.

3 Queue (FIFO – First In, First Out)

Definition:
A queue is a **linear data structure** where the **first element added is the first to be removed** (FIFO).

Real-World Analogy:
Think of a **queue at a ticket counter** – the first person in the queue gets served first.

Characteristics:

- Used in **task scheduling, message passing, event handling**.
- Elements are **added at the back** and **removed from the front**.

Example in JavaScript (Queue)

```
let queue = [];
queue.push("Customer 1");
queue.push("Customer 2");
queue.push("Customer 3");

console.log(queue.shift()); // Output: Customer 1
console.log(queue.shift()); // Output: Customer 2
console.log(queue.shift()); // Output: Customer 3
```

Queue in the Event Loop:
In JavaScript, the **Event Loop** processes tasks in a queue, ensuring **non-blocking execution**.

Example:

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

setTimeout(() => console.log("Timeout callback"), 0);

console.log("End");
```

Output:

```
Start
End
Timeout callback
```

Why?

- `console.log("Start")` and `console.log("End")` execute **synchronously**.
- `setTimeout()` adds a task to the **callback queue**, which runs **after the main script finishes**.

Summary Table

Feature	Stack	Heap	Queue
Order	LIFO (Last In, First Out)	No specific order	FIFO (First In, First Out)
Speed	**Fast**	**Slower** (because of dynamic allocation)	**Moderate**
Use Case	Function calls, local variables	Objects, dynamic memory	Asynchronous tasks, scheduling
Memory Management	Automatic (popped when function returns)	Manual (needs garbage collection)	Tasks removed when executed

When to Use Stack, Heap, and Queue?

Stack: When you need **fast execution** and automatic memory management (e.g., function calls).
Heap: When you need **flexible memory allocation** (e.g., objects, data persistence).
Queue: When handling **asynchronous tasks** (e.g., request handling, scheduling).

Asynchronous Programming with Callbacks in Node.js

Asynchronous programming allows tasks to execute **without blocking** the execution of other operations. This is crucial in **Node.js**, as it runs on a single-threaded event loop.

1 What is a Callback?

A **callback** is a function passed as an argument to another function and is executed after the operation completes.

- **Example: Synchronous vs. Asynchronous**

  - *Synchronous Code (Blocking)*

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

function syncFunction() {
    console.log("Executing sync task...");
}

syncFunction();

console.log("End");
```

- **Output** (Executed in order):

```
Start
Executing sync task...
End
```

- **Problem?** The task blocks the execution of further code.

2 Asynchronous Callbacks

In **asynchronous programming**, tasks **do not block execution**. The **callback** function runs after the asynchronous operation completes.

Example: Asynchronous Code with Callbacks

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

setTimeout(() => {
    console.log("Executing async task...");
}, 2000); // Executes after 2 seconds

console.log("End");
```

- **Output (Non-blocking execution)**:

```
Start
End
Executing async task...
```

- **Why?** `setTimeout` is asynchronous, so it schedules execution after 2 seconds but allows other code to continue running.

3 Callbacks in File System (Real Example)

In Node.js, file operations use asynchronous callbacks.

Example: Reading a File Asynchronously

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

console.log("Start");

fs.readFile('file.txt', 'utf8', (err, data) => {
    if (err) {
        console.log("Error reading file:", err);
        return;
    }
    console.log("File content:", data);
});
```

```
console.log("End");
```

◇ **Expected Output (Non-blocking execution)**:

```
Start
End
File content: (file contents appear here)
```

◇ **Why?** Node.js does not wait for `fs.readFile()` to complete; it moves on and executes the callback once the file is read.

4 Callback Hell (The Pyramid of Doom)

When multiple callbacks are nested, it leads to **callback hell**, making the code hard to read and maintain.

◇ Example: Callback Hell

```
fs.readFile('file1.txt', 'utf8', (err, data1) => {
    if (err) return console.log(err);

    fs.readFile('file2.txt', 'utf8', (err, data2) => {
        if (err) return console.log(err);

        fs.readFile('file3.txt', 'utf8', (err, data3) => {
            if (err) return console.log(err);

            console.log("All files read successfully!");
        });
    });
});
```

◇ **Problem?** 😵 The code is deeply nested and hard to manage.

Solution?

Use **Promises** or **Async/Await** instead of callbacks to avoid this problem.

5 Summary

☑ **Callbacks** enable asynchronous execution.
☑ Avoid **callback hell** using **Promises** or **Async/Await**.
☑ Use **error-first callbacks** (`(err, data) => {}`) in Node.js.

Promises in JavaScript

Promises are a **modern approach to handling asynchronous operations** in JavaScript. They help manage operations like fetching data, reading files, or executing timers without blocking the main thread.

1 What is a Promise?

A **Promise** is an object that represents the eventual **completion (success) or failure** of an asynchronous operation.

◇ Basic Syntax

```
let promise = new Promise((resolve, reject) => {
    // Asynchronous operation
    let success = true; // Change to false to simulate failure

    setTimeout(() => {
        if (success) {
            resolve("Operation Successful!"); // When successful
        } else {
            reject("Operation Failed!"); // When failed
        }
    }, 2000); // Simulating delay
});

promise
    .then((result) => console.log(result))  // Handle success
    .catch((error) => console.log(error));  // Handle failure
```

`resolve()` is called if the operation is successful.
`reject()` is called if the operation fails.

2 Promise States

A Promise can have **three states**:

State	Description
Pending	Initial state (operation not completed yet)
Fulfilled	Operation completed successfully (`resolve()`)
Rejected	Operation failed (`reject()`)

3 Chaining Promises

Promises allow **chaining** to execute multiple asynchronous operations **sequentially**.

Example: Fetching Data in Sequence

```
new Promise((resolve) => {
    setTimeout(() => resolve("Step 1 Complete"), 1000);
})
    .then((result) => {
        console.log(result);
        return new Promise((resolve) => setTimeout(() => resolve("Step 2
Complete"), 1000));
    })
    .then((result) => {
        console.log(result);
        return new Promise((resolve) => setTimeout(() => resolve("Step 3
Complete"), 1000));
    })
    .then((result) => console.log(result))
    .catch((error) => console.log("Error:", error));
```

Each **`.then()`** receives the previous Promise's result and returns a new Promise.
If any step **fails**, the `.catch()` block handles the error.