Server-Side Development using Node JS

What is Node JS?

- Node.js is a free, open source tool that lets you run ***JS*** outside the web browser.
- With Node.js, you can build fast and scalable applications like web servers, APIs, tools, and more.
- Node.js uses an event-driven, non-blocking model.
- It can handle many connections at once without waiting for one to finish before starting another.
- This makes it great for real-time apps and high-traffic websites.

What is Node JS?

- Here are some examples of what you can build with Node.js:

➢Web servers and websites
➢APIs
➢Real-time apps
➢Command-line tools
➢Working with files and databases
➢IoT and hardware control

- Coding Example

Advantages of Node JS

1. Asynchronous & Non-Blocking I/O

- Node.js uses an event-driven, non-blocking I/O model.
- This makes it lightweight and efficient, especially for data-intensive real-time applications like chats, streaming services, or APIs.

2. High Performance

- Built on Google's JavaScript engine, Node.js compiles JavaScript directly into machine code.
- It can handle thousands of concurrent connections with high throughput, making it suitable for scalable applications.

Advantages of Node JS

3. Single Programming Language (JavaScript Everywhere)

- Developers can use JavaScript for both **frontend and backend**, reducing the context-switching between languages.
- This unification improves productivity and makes full-stack development simpler.

4. Large Ecosystem (NPM)

- Node.js has one of the largest open-source package ecosystems—**npm (Node Package Manager)**.
- Developers can easily integrate pre-built libraries, reducing development time and effort.

Advantages of Node JS

5. Real-Time Application Support

- Perfect for apps requiring real-time interactions:

➢Chat applications

➢Online gaming

➢Live streaming

➢Collaborative tools (Google Docs-like apps)

Advantages of Node JS

6. Scalability

- Node.js applications can scale both **vertically** (adding resources to a single server) and **horizontally** (adding more servers).
- Its microservices-friendly nature supports distributed systems.

7. Cross-Platform Development

- Frameworks like **Electron** and **NW.js** allow building cross-platform desktop apps.
- With **React + Node.js**, developers can build mobile apps as well.

Advantages of Node JS

8. Community Support

- Node.js has a huge and active developer community.
- Regular updates, tutorials, libraries, and frameworks make it developer-friendly.

9. JSON Support

- Node.js works seamlessly with JSON (JavaScript Object Notation).
- This makes it excellent for building APIs, interacting with NoSQL databases (like MongoDB), and handling structured data.

Advantages of Node JS

10. Cost Efficiency

- Since one language (JavaScript) can be used across the stack, fewer developers are required for backend/frontend separation.

- This reduces development and hiring costs.

Traditional Web Server Model

1. Request–Response Cycle

- A client (browser) sends a request to the server (e.g., loading a webpage).
- The server processes the request, often involving:
  - Parsing the request
  - Running server-side scripts (PHP, Java Servlets etc.)
  - Querying a database
  - Generating an HTML page
- The server sends back the response to the client.

Traditional Web Server Model

2. Thread/Process-Based Model

- Traditional servers often use a **multi-threaded** or **multi-process** model.
- For each incoming client request:
  - A new thread (or process) is created, or
  - A thread from a thread pool is assigned.
- Each thread handles one request until it’s completed.

Traditional Web Server Model

3. Blocking I/O

- Input/Output operations (like file access or database queries) are usually **blocking**.
- This means the thread handling the request is paused until the operation completes.
- While scalable, this can become resource-heavy when thousands of requests come in.

Traditional Web Server Model

4. Scalability Characteristics

- Suitable for **moderate traffic** web applications.
- With very high concurrency, performance suffers because:
  - Each thread consumes memory and CPU.
  - Context switching between threads adds overhead.
  - Risk of server crashes under load.

Traditional Web Server Model

5. Advantages

- **Mature and stable**: Battle-tested with decades of use.
- **Rich ecosystem**: Many frameworks (Django, Laravel, etc.).
- **Easy debugging**: Thread-per-request model is straightforward.
- **Strong tooling**: Well-supported with IDEs, monitoring tools, and enterprise integrations.

Traditional Web Server Model

6. Disadvantages

- **High resource usage**: Each thread consumes significant memory.
- **Scalability bottleneck**: Performance drops with massive concurrent connections.
- **Slower real-time handling**: Not well-suited for apps requiring persistent connections (chat, streaming).
- **Latency**: Blocking I/O can increase response times.

Node.js Process Model (Architecture)

- Node.js uses a **single-threaded, event-driven** architecture that is designed to handle many connections at once, efficiently and without blocking the main thread.

- This makes Node.js ideal for building scalable network applications, real-time apps, and APIs.

Node.js Process Model (Architecture)

- A simple overview of how Node.js processes requests:

1. Client Request Phase

- Clients send requests to the Node.js server
- Each request is added to the **Event Queue**

2. Event Loop Phase

- The Event Loop continuously checks the **Event Queue**
- Picks up requests one by one in a loop

Node.js Process Model (Architecture)

3. Request Processing

- Simple (non-blocking) tasks are handled immediately by the main thread
- Complex/blocking tasks are offloaded to the Thread Pool

4. Response Phase

- When blocking tasks complete, their callbacks are placed in the **Callback Queue**
- Event Loop processes callbacks and sends responses

Node.js Process Model (Architecture)

- Non-Blocking Nature (Coding Example)

** Now, if you noticed carefully how "After file read" is printed before the file contents, showing that Node.js does not wait for the file operation to finish.*

- So, to conclude we can say Node.js is particularly well-suited for:

- **I/O-bound applications** - File operations, database queries, network requests
- **Real-time applications** - Chat apps, live notifications, collaboration tools
- **APIs** - RESTful services (Secure information exchange)
- **Microservices** - Small, independent services

NodeJS : Environment Setup

1. Download and Install Node.js
   (Go to https://nodejs.org)

2. Verify Installation
   Open your terminal/command prompt and type:
   “node --version” and “npm --version”

You should see version numbers for both Node.js and npm

3. Once you have installed Node.js, you may create your first server that in a web browser.

NodeJS : Environment Setup

4. Once created you can initiate by typing :

“node [name of the file]”

5. If above steps are executed correctly you would have initiated your server which can be accessed in the browser by typing the address “http://localhost:8080”

Node JS Console

- The console module is essential for debugging and logging in Node.js applications.
- It enables developers to print messages to the terminal, making it easier to monitor application behavior, track issues, and display runtime information.
- The console module in Node.js is a built-in utility that provides access to the standard output and error streams, offering various methods for printing information, debugging, and logging messages.

Node JS Console

Console Methods

- **console.log():** This method is used to output general information or debugging messages to the console.
- **console.error():** This method is used to display error messages in the console.
- **console.warn():** This method is used to display warning messages in the console.
- **console.count():** It is used to count the number of times a specific label has been called.
- **console.clear():** It is used to clear the console history.
- **console.info():** It is used to write a messages on console and it is an alias of console.log() method.
- **console.time():** It is used to get the starting time of an action.

Node JS Modules

- Modules are the building blocks of Node.js, allowing you to organize code into logical, reusable components. They help in:

➢Organizing code into manageable files

➢Encapsulating functionality

➢Preventing global namespace pollution

➢Improving code maintainability and reusability

Node JS Modules

- Node.js provides several built-in modules that are compiled into the binary.
- To use any built-in module, the require() function is used.

`fs` - File system operations	`util` - Utility functions
`http` - HTTP server and client	`stream` - Stream handling
`path` - File path utilities	`crypto` - Cryptographic functions
`os` - Operating system utilities	`url` - URL parsing
`events` - Event handling	`querystring` - URL query string handling

Node JS Modules

- In Node.js, any file with a .js extension is a module. You can export functionality from a module in several ways.

- **Exporting Multiple Items** : Add properties to the exports object for multiple exports.
- **Exporting a Single Item**: To export a single item (function, object, etc.), assign it to module.exports.
- **Using Custom Modules**: Import and use your custom modules using require().

Node JS Modules

- **Note:** Node.js caches the modules after the first time they are loaded. This means that subsequent require() calls return the cached version.

Node JS Modules Types

➢Core Modules

- Preloaded with Node.js installation.
- No need to install separately.
- Examples:

❖**http** → Create web servers.
❖**fs** → File system operations (read/write).
❖**path** → Work with file and directory paths.
❖**os** → Provides system-related information.

Node JS Modules Types

➢**Core Module** (fs) – Coding Example

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

Node JS Modules Types

➢ **Local Modules (User-defined)**

- Created by developer for specific project functionality.
- These are custom JavaScript files written inside the project.

Node JS Modules Types

➢**Local Modules – Coding Example**

```
// math.js
exports.add = (a, b) => a + b;
exports.sub = (a, b) => a - b;

// app.js
const math = require('./math');
console.log(math.add(10, 5)); // 15
```

Node JS Modules Types

➢**Third-party Modules**

- Installed via npm (Node Package Manager).
- Extends functionality.

- Examples:

❖**express** → Web application framework.
❖**mongoose** → MongoDB object modeling.
❖**lodash** → Utility functions.

Node JS Modules Types

➢**Third-party Modules (express) – Coding Example**

```
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'));
```

Node JS Modules Types

➢Global Modules

- Accessible without require().

- Available globally in Node.js.

- Examples:

❖**console** → Logging information.

❖**process** → Provides information about the current process.

❖**Buffer** → Handles binary data.

Node JS Modules Types

- **Global Modules – Coding Example**

console.log('Node Version:', process.version);

Node JS Functions

- **Functions** are blocks of reusable code that developer can define and call to perform specific tasks.

- They can be synchronous or asynchronous and can be declared in multiple ways.

- They may be used for creating server-side applications, command-line tools, and other JavaScript-based utilities outside of a web browser.

Node JS Functions

- **Key aspects of Functions:**

- Functions in Node.js follow the same syntax **as *JavaScript* functions**, utilizing the ”*function*” keyword, parameters, and a function body enclosed in curly braces.

- Example

```
function greet(name)
{    console.log(`Hello, ${name}!`);
}
```

Node JS Functions

- Developer can define functions using both declarations and expressions.

- Example

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

 // Function Expression
const subtract = function(a, b)
{    return a - b;
 }
```

Node JS Functions

- Functions create their own scope, meaning variables declared inside a function are not accessible from outside.
- Node.js heavily relies on asynchronous operations and event-driven architecture.
- Functions often accept callback functions as arguments to handle the results of asynchronous tasks, like file I/O or network requests.
- Functions can be organized into modules and exported using *”module.exports”* to make them available for use in other files, thereby resulting in code reusability and maintainability.

Node JS Functions

- Synchronous vs Asynchronous Functions

Synchronous	Asynchronous
• Blocks execution until complete • Simple to understand • Can cause delays • Uses functions like *”readFileSync”*	• Non-blocking execution • Better performance • More complex to handle • Uses callbacks and or async/await.

Node JS Functions

- Coding Example

➢Synchronous File Read

➢Output will be in order: 1 → 2 → 3 (blocks between each step)

➢Asynchronous File Read

➢Output order: 1 → 3 → 2 (doesn't wait for file read to complete)

Node JS Functions

- Synch vs Asynch Functions (Advantages and Disadvantages)

- **Advantages**

➢ Use async/await for better readability.
➢ Always handle errors with try/catch.
➢ Run independent operations in parallel.

- **Disadvantages**

➢ Mixing up of sync and async code does not work well.
➢ Nesting of callbacks makes it hard to read and backtrack.

Node JS Functions

- Note : Asynchronous code lets Node.js handle many requests at once, without waiting for slow operations like file or database access. This makes Node.js great for servers and real-time apps.

Nodejs Buffer

- One of the core modules under Nodejs. Mainly used to handle binary data.

- They are similar to arrays of integers but are fixed in size and represent raw memory allocations.

- Coding Example

Nodejs Buffer

- There are several ways to create buffers in Nodejs.
- **Buffer.alloc():** Creates a new Buffer of the specified size, initialized with zeros.
- **Buffer.allocUnsafe():** Creates a new Buffer of the specified size, but doesn't initialize the memory.
- **Buffer.from():** Creates a new Buffer from various sources like strings, arrays, or ArrayBuffer.
- **Buffer.compare():** Compares two buffers and returns a number indicating whether the first one comes before, after, or is the same as the second one

Nodejs Buffer

- **buffer.copy():** Copies data from one buffer to another.
- **buffer.slice():** Creates a new buffer that references the same memory as the original
- **buffer.toString():** Decodes a buffer to a string.

Extra : Nodejs Encodings

- Supported encodings in Node.js include:

- **utf8**: Multi-byte encoded Unicode characters (default)
- **ascii**: ASCII characters only (7-bit)
- **latin1**: Latin-1 encoding (ISO 8859-1)
- **base64**: Base64 encoding
- **hex**: Hexadecimal encoding
- **binary**: Binary encoding (deprecated)
- **ucs2/utf16le**: 2 or 4 bytes, little-endian encoded Unicode characters

Nodejs : Event Driven Architecture

- Node.js is **built on an event-driven architecture** — this is the reason it can handle thousands of concurrent connections efficiently without creating a new thread for each one.

- Instead of running tasks sequentially and waiting for blocking operations (like file read, database query, network call), Node.js:

➢Registers **listeners** (functions) for certain **events**.

➢When an event occurs Node.js triggers the associated callback.

➢This is powered by the **Event Loop** and the *Event-Emitter* pattern.

Nodejs : Event Driven Architecture

- Core components of Nodejs event driven architecture

A. Event Loop

Acts as the heart of Nodejs. Keeping the pending events in check and dispatches callbacks. Helps in avoiding blocking of threads.

B. Event Emitter

Built in Nodejs events class which lets the developer create, emit and listen for custom events.

Nodejs : Event Driven Architecture

C. Non-blocking I/O

Node uses asynchronous system calls so the main thread doesn't wait.

- **Advantages of event driven architecture :**

➢Highly Scalable
➢Fast
➢Lightweight

Nodejs : Event Driven Architecture

- **Disadvantages**

➢ May develop nesting of call back resulting in code mess.

➢ CPU intensive tasks may lead to event loop block.

➢ Error Handling if not done correctly may lead to unmanageable code

Node Package Manager (NPM)

- Node.js ships with **Node Package Manager (NPM) —** the world's largest software registry for JavaScript libraries and tools.
- It's an essential part of Node.js development for sharing and managing code.

- NPM mainly is two things :

a. **Online Registry** - a huge public database of JavaScript packages
b. **CLI Tool** - a command-line interface installed with Node.js to install, publish, and manage packages.

Node Package Manager (NPM)

- NPM can be used for :

➢ **Installing Packages**: Reuse existing libraries instead of writing from scratch.

➢ **Dependency Management**: Handles versioning and nested dependencies automatically.

➢ **Scripts**: Run build, test, deploy tasks.

➢ **Publish Packages**: Share your own code with others.

➢ **Security Audits**: Scan and fix vulnerabilities (*npm audit*).

Node Package Manager (NPM)

- NPM comes automatically with Node.js.

➢ *node -v* → Check Node.js version

➢ *npm -v* → Check NPM version

Node Package Manager (NPM)

- Popular NPM Packages:

- **Express** — web framework
- **Mongoose** — MongoDB ODM
- **Nodemon** — auto-restart server
- **Jest/Mocha** — testing
- **axios** — HTTP client