Introduction to Node JS & Node JS Modules

Node.js Basics

What is Node.js and what is it used for?

Node.js is a runtime environment that allows you to run JavaScript code outside of a browser. It is built on Chrome's V8 JavaScript engine and provides an event-driven, non-blocking I/O model, making it efficient and suitable for building scalable network applications.

Node.js is commonly used for

- Developing server-side applications.
- Enabling developers to use JavaScript for both front-end and back-end development.
- Helps in real-time communication, such as chat applications, live streaming, and online gaming.
- Used in creating RESTful APIs and GraphQL APIs.
- Building microservices, enabling developers to break down a large application into smaller, manageable services that can be developed, deployed, and scaled independently.

Explain the main differences between Node.js and traditional web server

Node.js	Traditional web server
Uses a single-threaded, event-driven architecture. It employs an event loop to handle asynchronous operations, allowing it to manage many connections simultaneously without blocking the main thread.	Typically use a multi-threaded or multi-process model. Each incoming request is handled by a separate thread or process, which can be more resource-intensive, especially under heavy load.
Uses non-blocking, asynchronous I/O operations, which means it can handle multiple I/O operations concurrently. This leads to better performance and scalability for I/O-intensive applications.	Traditionally use blocking, synchronous I/O operations. While modern versions can support asynchronous handling, they generally rely more on concurrent threads or processes.
Uses JavaScript for both server-side and client-side development, enabling full-stack development with a single language.	Can work with various programming languages like PHP, Python, Ruby, and others. This requires developers to use different languages for server-side and client-side development.

Ideal for real-time applications (e.g., chat applications, live notifications), single-page applications (SPAs), API servers, and microservices due to its non-blocking nature and efficient handling of concurrent connections.	Well-suited for serving static content, handling HTTP requests, and acting as a reverse proxy or load balancer. They are known for their stability and extensive configuration options.

What is the V8 engine and how does Node.js utilize it?

The V8 engine is an open-source JavaScript engine developed by Google. It is written in C++ and is used in Google Chrome to execute JavaScript code. The V8 engine compiles JavaScript directly to native machine code, providing high performance and efficiency.

Node.js utilize V8 engine to

- Execute JavaScript code on the server side. V8 compiles the JavaScript code into machine code that can be directly executed by the computer's processor, allowing for fast and efficient execution.
- Leverages these optimizations to handle high-throughput and low-latency applications effectively, because the V8 engine is designed for high performance.
- Manage memory allocation and deallocation, because it avoids memory leaks.

Describe the event-driven architecture of Node.js.

Node.js's event-driven architecture revolves around an event loop, enabling non-blocking I/O operations. When an I/O task (like reading a file or making a network request) starts, Node.js doesn't wait for it to complete; instead, it continues executing other code. Once the task finishes, a callback function or a promise handles the result. The EventEmitter class allows objects to emit events and register listeners. This approach efficiently manages multiple concurrent connections with minimal resource use, making Node.js highly performant, scalable, and suitable for real-time and I/O-bound applications. For example:

E.g.

```
const fs = require('fs');
fs.readFile('example.txt', 'utf8', (err, data) => {
  if (err) {
    console.error('Error reading file:', err);
    return;
  }
  console.log('File content:', data);
});
console.log('This log happens while the file is being read');
```

What are some common use cases for Node.js?

- Node.js is used to build fast and scalable web servers and APIs, leveraging its non-blocking I/O model for high concurrency.
- Ideal for applications requiring real-time updates, such as chat applications, gaming servers, and live tracking apps.
- Node.js serves as a backend for SPAs, working seamlessly with front-end frameworks like React, Angular, and Vue.js.
- Node.js facilitates building microservices architectures due to its lightweight nature and efficient handling of multiple services.
- Used in streaming applications and IoT for device communication, data processing, and monitoring.

How does Node.js handle asynchronous operations?

Node.js manages asynchronous operations through its event-driven, non-blocking architecture. When an asynchronous task, such as reading a file or making a network request, is initiated, Node.js delegates it to the system and continues executing other tasks without waiting for the operation to complete. Once the task finishes, Node.js uses callback functions, promises, or async/await syntax to handle the results. This approach allows Node.js to efficiently handle multiple concurrent operations, maximizing throughput and responsiveness in applications requiring real-time data processing and I/O-intensive tasks.

What is the purpose of the package.json file in a Node.js project?

The `package.json` file in a Node.js project serves as a manifest that provides essential information about the project and its dependencies. It includes metadata such as the project's name, version, description, author, and license. Additionally, it specifies the dependencies and devDependencies required for the project to run and develop correctly, ensuring consistent environments across different setups. The file also defines scripts for common tasks like starting the server, running tests, and building the project. Overall, `package.json` is crucial for managing and automating various aspects of a Node.js project.

Explain the role of the Node Package Manager (NPM).

- NPM allows developers to easily install, update, and manage libraries and dependencies needed for their Node.js projects. By running simple commands, developers can add or remove packages from their projects, and NPM takes care of handling dependencies and versioning.
- NPM hosts a vast repository of open-source Node.js packages, providing a centralized location for discovering and sharing reusable code. Developers can publish their own packages to the NPM registry, making them available for others to use.
- NPM helps manage different versions of packages, ensuring compatibility and stability in projects. It allows developers to specify the exact versions or version ranges of packages their project depends on, preventing issues caused by incompatible updates.

- NPM allows developers to define scripts in the '`package.json`' file to automate common tasks like testing, building, and deploying applications. These scripts can be run with simple NPM commands, streamlining the development workflow.

What is the node_modules folder and why is it important?

The '`node_modules`' folder in a Node.js project is a directory where all the installed dependencies and their respective dependencies are stored

Importance:

- It contains all the libraries and packages that the project depends on, as specified in the '`package.json`' file. This includes both direct dependencies and their nested dependencies.
- By storing dependencies locally within the project, `node_modules` ensures that each project has its own set of required libraries, preventing conflicts between different projects on the same machine.
- The `node_modules` folder works with the `package-lock.json` file to maintain consistent package versions across different environments. This ensures that the project behaves the same way on every developer's machine and in production.
- Having all dependencies in one place allows Node.js to quickly resolve and load modules when the application runs, improving performance and reducing the complexity of module resolution.
- The `node_modules` folder makes it easy for developers to add, update, or remove dependencies using NPM commands. The folder is automatically updated to reflect the changes, streamlining the development workflow.

How can you check the version of Node.js and NPM installed on your system?

To check the version of Node.js and NPM installed on your system, you can check by typing the codes in command prompt:

Check Node.js Version: node –version

Check NPM Version: npm –version

How does Node.js handle concurrency and what are the benefits of this approach?

How Node.js Handles Concurrency

- Event Loop
  - The event loop is the core mechanism that allows Node.js to handle multiple operations concurrently. It continuously checks for and processes events, such as I/O operations or timers, without blocking the main thread.
- Non-Blocking I/O

    - Node.js uses non-blocking I/O operations, meaning it initiates I/O tasks (like reading a file or querying a database) and then continues executing other code. Once the I/O operation completes, a callback function or a promise handles the result, allowing the main thread to remain free for other tasks.
- Asynchronous Callbacks and Promises
    - Asynchronous operations in Node.js are typically handled using callbacks, promises, or async/await. These mechanisms allow Node.js to manage tasks that take time to complete (like network requests) without waiting and blocking the main thread.
- Worker Threads
    - For CPU-intensive tasks, Node.js provides a Worker Threads module that allows execution of JavaScript in parallel threads. This helps offload heavy computations from the main event loop, maintaining responsiveness.

Benefits of This Approach

- High Performance
    - By avoiding the need for multiple threads and context switching, Node.js can handle a large number of simultaneous connections with minimal overhead, making it highly performant for I/O-bound applications.
- Scalability
    - Node.js's non-blocking, event-driven model scales well with increased load, as it efficiently manages multiple concurrent operations without consuming excessive system resources.
- Responsiveness
    - The non-blocking nature ensures that the application remains responsive, as the main thread is never blocked by long-running operations. This is especially beneficial for real-time applications like chat apps and online games.

- Simplified Development
    - Writing code for single-threaded execution can be simpler and less error-prone compared to multi-threaded programming, which often involves complex issues like deadlocks and race conditions.
- Resource Efficiency
    - Node.js's single-threaded model consumes less memory and CPU compared to traditional multi-threaded servers, which spawn multiple threads or processes to handle concurrent connections.

How does Node.js handle file I/O? Provide an example of reading a file asynchronously.

Node.js handles file I/O using its `fs` (file system) module, which provides asynchronous methods to perform non-blocking file operations. For example, to read a file asynchronously, you can use `fs.readFile('example.txt', 'utf8', (err, data) => { if (err) { console.error('Error reading file:', err); return; } console.log('File content:', data); });`. This approach allows Node.js to initiate the file read operation and immediately continue executing other code. When the file read operation completes, the provided

callback function handles the result, ensuring the main thread remains free and responsive during the I/O operation.

Node.js Modules

What are streams in Node.js and how are they useful?

- **Key Types of Streams:**

- Readable Streams: Used to read data from a source. Examples include reading files, HTTP responses, and standard input.
- Writable Streams: Used to write data to a destination. Examples include writing files, HTTP requests, and standard output.
- Duplex Streams: Both readable and writable, allowing for bidirectional communication. Examples include network sockets.
- Transform Streams: A type of duplex stream where the output is computed based on the input, such as compression or encryption.

- **How Streams Are Useful:**

- Memory Efficiency: Streams process data in chunks, which is efficient for handling large files or data transfers, as it avoids loading the entire dataset into memory.
- Time Efficiency: Streams start processing data as soon as it begins to arrive, providing a more responsive and faster approach compared to waiting for the entire dataset to be available.
- Piping: Streams can be easily piped together using the .pipe() method, creating a chain of processing steps. For example, reading data from a file and directly writing it to another file or a network socket.
- Flexibility: Streams can handle various types of data sources and destinations, making them versatile for a range of applications like file manipulation, network communication, and real-time data processing.

What are modules in Node.js and why are they important?

Modules in Node.js are JavaScript files that encapsulate functionality, making it reusable and maintainable across different parts of an application. They help organize code into smaller, focused units, enhancing readability and facilitating collaboration among developers. Modules also support code reusability by allowing functions, classes, or variables defined within them to be imported and used in other modules or files. This modular structure is essential for building complex applications, enabling developers to manage dependencies, avoid naming conflicts, and promote code separation, thereby improving overall code quality and scalability.

How do you create a module in Node.js? Provide a simple example.

define functionality within a JavaScript file and export it using `module.exports`

eg:

```
// Function to add two numbers
const add = (a, b) => {
  return a + b;
};
// Function to multiply two numbers
const multiply = (a, b) => {
  return a * b;
};
// Exporting functions to be used as a module
module.exports = {
  add,
  multiply
};
```

Explain the difference between require and import statements in Node.js.

In Node.js, `require` is the traditional way to import modules using CommonJS syntax, like `const module = require('module')`. It's resolved synchronously at runtime and supports importing entire modules or specific parts (`exports`). On the other hand, `import` is part of the ECMAScript module syntax (`ESM`) and uses declarative statements (`import module from 'module'` or `import { namedExport } from 'module'`). It's resolved statically at compile-time and is increasingly supported in Node.js with proper configuration (`"type": "module"` in `package.json`).

What is the module.exports object and how is it used?

In Node.js, `module.exports` is a special object used to define what a module exports, making its functionalities available for other modules to require and use. By assigning properties or functions to `module.exports`, you can export them for consumption elsewhere in your application. This mechanism is essential for organizing and reusing code across different parts of a Node.js project, ensuring clean and modular code architecture. Would you like me to remember these details about modules and `module.exports` for future conversations?

Describe how you can use the exports shorthand to export module contents.

In Node.js, `exports` is a shorthand alias for `module.exports` that allows you to directly assign properties or functions to export from a module. By using `exports`, you simplify the syntax for exporting functionalities without needing to explicitly assign to `module.exports`. This shorthand is particularly useful for exporting multiple functions or variables within a single module, facilitating clean and concise code organization.

What is the CommonJS module system?

The CommonJS module system is a module format for JavaScript used primarily in server-side environments like Node.js. It provides a mechanism to structure code into reusable modules using `require` to import modules and `module.exports` to export values. Modules are loaded synchronously, ensuring dependencies are resolved before execution. This system facilitates modular development, making it easier to manage dependencies and organize code in a clear and maintainable manner within Node.js applications.

How can you import a module installed via NPM in your Node.js application?

To import a module installed via NPM in Node.js, use require followed by the module's name. For instance, to import the lodash module:

Eg: const _ = require('lodash');

This allows you to access and use functions provided by the lodash module within your application.

Explain how the path module works in Node.js. Provide an example of using it.

The path module in Node.js provides utilities for working with file and directory paths. For example, to join paths:

Eg: const path = require('path');
const fullPath = path.join(__dirname, 'file.txt');

How do you handle circular dependencies in Node.js modules?

Circular dependencies occur when two or more modules require each other directly or indirectly. To handle this, structure your code to minimize interdependence, use techniques like delayed evaluation, or refactor code to remove circular dependencies.

What is a built-in module in Node.js? Name a few and explain their purposes.

Built-in modules are part of Node.js core and include fs (file system) for file operations, http for creating HTTP servers, and util for utility functions. They provide essential functionalities to interact with the operating system and network.

What is the difference between relative and absolute module paths in Node.js?

Relative paths ('./module') refer to modules located relative to the current file, while absolute paths ('/module') refer to modules located based on the root directory of the file system. Relative paths are more portable, while absolute paths provide explicit references to module locations.

What is a module wrapper function in Node.js?

The module wrapper function is an encapsulating function that surrounds every module in Node.js before its code is executed. It provides a scope for the module's code and includes several variables and functions such as exports, require, module, __filename, and __dirname. This function ensures that each module's code is isolated and can safely interact with other modules using the CommonJS module system.

Describe the buffer module and its use in Node.js.

The buffer module in Node.js provides a way to handle binary data, especially when working with streams, file systems, or network protocols that involve raw data transmission. It allocates fixed-size blocks of memory (buffers) to efficiently store and manipulate binary data, which may include images, files, or network packets. Buffers are used to perform operations like reading and writing data, converting between different encoding formats (e.g., UTF-8, Base64), and handling binary data in a structured manner within Node.js applications.

Starting an HTTP Server in Node.js

How do you create a simple HTTP server in Node.js? Provide a code example.

```
const http = require('http');
const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello World!\n');
});
server.listen(3000, () => {
  console.log('Server is running on http://localhost:3000');
});
```

Explain the purpose of the http module in Node.js.

The http module in Node.js provides functionality to create HTTP servers and handle HTTP requests and responses. It facilitates building web applications, APIs, and services by enabling communication over the HTTP protocol.

What method do you use to start the HTTP server and make it listen on a specific port?

By using the listen method on the server object to start the HTTP server and make it listen on a specific port.

Eg:

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

How can you send a response to the client in an HTTP server created with Node.js?

Use the res object's methods (writeHead to set headers, end to send content) to send a response.

Eg:

```
res.writeHead(200, { 'Content-Type': 'text/plain' });
res.end('Hello World!\n');
```

Explain the request and response objects in the context of an HTTP server.

- **Request** (req): Represents the HTTP request from the client, containing information such as URL, HTTP method, headers, and body.
- **Response** (res): Represents the HTTP response that the server sends back to the client, including headers and content.

How do you handle different HTTP methods (GET, POST, etc.) in a Node.js HTTP server?

Use conditional statements or routing frameworks (if statements or switch statements) based on req.method to handle different HTTP methods.

Eg:

```
if (req.method === 'GET') {
 // Handle GET request
} else if (req.method === 'POST') {
```

```
 // Handle POST request
}
```

What is middleware in the context of a Node.js HTTP server?

Middleware are functions that have access to the request (req) and response (res) objects in an HTTP server's request-response cycle. They can modify these objects, execute code, and terminate the request-response cycle, or pass control to the next middleware function. Middleware are used for tasks like logging, authentication, parsing request bodies, etc.

How can you serve static files using an HTTP server in Node.js?

Use the express framework or fs module to read and serve static files.

Eg:

```
const fs = require('fs');
const http = require('http');
const server = http.createServer((req, res) => {
  fs.readFile(__dirname + '/public/index.html', (err, data) => {
    if (err) {
      res.writeHead(404);
      res.end(JSON.stringify(err));
      return;
    }
    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end(data);
  });
});
server.listen(3000, () => {
  console.log('Server is running on http://localhost:3000');
});
```

Explain how to handle errors in an HTTP server created with Node.js.

Use try-catch blocks for synchronous code and error-first callbacks for asynchronous operations. You can also use middleware or error handling functions to catch and handle errors globally or per request.

How can you implement routing in a Node.js HTTP server without using external libraries?

Implement routing using conditional statements (if-else or switch-case) based on req.url or req.method.

Eg:

```
const server = http.createServer((req, res) => {
  if (req.url === '/' && req.method === 'GET') {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('Hello World!\n');
  } else if (req.url === '/about' && req.method === 'GET') {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('About Us\n');
  } else {
    res.writeHead(404, { 'Content-Type': 'text/plain' });
    res.end('Not Found\n');
  }
});
```