GPT- https://chatgpt.com/c/67382f62-5c50-8012-a6fe-5a2680f27dad

Q1. What is Node.js?

Node.js is a runtime environment that allows you to run JavaScript code on the server side, outside of the browser. With Node.js, JavaScript can be used to write backend code, manage server operations, interact with databases, handle HTTP requests, and more, all using a single language across both client and server sides.

Node.js is perfect for data-intensive applications as it uses an asynchronous, event-driven model. You can use I/O intensive web applications like video streaming sites. You can also use it for developing: Real-time web applications, Network applications, General-purpose applications, and Distributed systems.

A **data-intensive application** is an application that handles and processes large volumes of data, where the focus is on managing, storing, retrieving, and analyzing data rather than on complex computations or CPU-intensive tasks.

Examples of data-intensive applications include:

- **Social Media Platforms**
- **E-commerce Websites**
- **Streaming Services**
- **Big Data Analytics Platforms**
- **Financial Systems**

By definition, Node.js is an open-source and cross-platform JavaScript runtime environment that runs based on Chrome's V8 engine.

To break this down a little further, you can:

- Quickly find the source code for Node.js (open source).
- Run Node.js on Linux, Windows, or macOS (cross-platform).
- Execute your JavaScript program or application on the server instead of the browser using Node.js (runtime environment).

Q2. What is a REST API?

REST, which stands for Representational State Transfer, is a software development architecture that defines a set of rules for communication between a client and a server.

A RESTful API is a web service that follows the principles of REST architecture. It uses standard HTTP methods (GET, POST, PUT, DELETE) to perform CRUD (Create, Read, Update, Delete) operations on resources, and data is typically transferred in JSON or XML format.

Let's break this down a little more:

- A **REST client** is a code or app used to communicate with REST servers.
- A **server** contains resources that the client wants to access or change.
- A **resource** is any information that the API can return.

JSON (JavaScript Object Notation)

- **Format**: Lightweight and human-readable, uses key-value pairs.
- **Structure**: Relies on arrays and objects.
- **Usage**: Commonly used in web APIs for data exchange.
- **Supported Data Types**: String, Number, Boolean, Null, Array, Object.

- **Advantages**:
    - Easier to read and write.
    - Consumes less bandwidth due to smaller size.
    - Directly parseable in JavaScript (and many other programming languages).

❖ **Node js can be used for both frontend as well as backend.**

Q3. Nodejs is asynchronous what does that mean.

When we say Node.js is asynchronous, it means that Node.js allows multiple operations to occur in parallel, without blocking the execution of other tasks. In other words, it doesn't wait for one operation (like reading a file from disk or querying a database) to complete before moving on to the next operation.

Here's how it works:

- **Non-blocking I/O:** Node.js uses non-blocking input/output (I/O) operations. When Node.js performs a task like reading a file or making an HTTP request, it doesn't pause the program until the task is complete. Instead, it initiates the task and moves on to the next operation, allowing the application to handle other tasks.
- **Event-driven architecture:** Node.js relies on an event-driven model. When the asynchronous task (e.g., file reading) completes, it triggers an event (like a callback function or a promise resolution), letting the program know that the operation is done and providing the result.

Example:

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

// Asynchronous operation
fs.readFile('example.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data);
});
console.log('This will run before the file is read.');
```

In the above code:

- fs.readFile is an asynchronous operation. It starts reading the file, but it doesn't stop the execution of the program.
- The console.log('This will run before the file is read.') will execute before the file reading operation completes because Node.js moves on to the next operation immediately.

This non-blocking behavior allows Node.js to handle a large number of operations efficiently, especially in I/O-heavy applications like web servers.

Key Concepts of Execution:

1. **Event Loop**: Node.js uses an event loop to manage asynchronous operations. When an async operation starts, the program continues executing subsequent code while the async task runs in the background.
2. **Non-blocking I/O**: Asynchronous functions don't block the execution of the rest of the code.

3. **Callback Execution**: Once the async operation completes, the callback function is added to the event loop's queue and executed.

Q4. what is event driven explain in details

Event-driven architecture is a programming paradigm where the flow of a program is determined by events — such as user actions (clicks, key presses), sensor outputs, or messages from other programs. In this architecture, events are the core component that trigger changes and responses in the system. It decouples the producer of events from the consumers, making it a flexible and scalable way to handle complex systems, especially when real-time responses are needed.

Event-driven architecture is a powerful programming model where actions (events) dictate the flow of the system. The system doesn't follow a rigid, step-by-step execution path, but instead responds dynamically to events that occur. By using event emitters, event handlers, and an event loop, systems can handle a large number of events efficiently and in real-time, making event-driven systems especially useful for real-time applications, GUIs, IoT, and scalable server architectures.

When we say **Node.js is event-driven**, it means that the execution of the program is largely controlled by events and event handlers. Node.js uses an event-driven architecture where the flow of the application is determined by events like user actions, server requests, file I/O completions, etc., and the corresponding functions (event handlers) that are triggered when those events occur.

Key Concepts in Event-Driven Architecture:

1. **Events**:
    - An event is an action or occurrence recognized by software that may be handled by event-handling code. Examples of events include user actions like mouse clicks, system-generated events like file I/O completion, network requests, or even timer expirations.
2. **Event Handlers (Listeners):**
    - An event handler or listener is a function or method that waits for a specific event to occur and executes a response when the event is detected. It listens to certain events and reacts accordingly when they are triggered.
3. **Event Emitters:**
    - An event emitter (sometimes called an event producer) is a part of the system that generates and broadcasts an event. For example, in Node.js, the EventEmitter class allows emitting named events that other parts of the code can listen to.
4. **Event Loop:**
    - The event loop is a core concept in event-driven programming. It continuously monitors and waits for events to occur, then dispatches them to appropriate event handlers. In systems like Node.js, the event loop allows non-blocking I/O operations, meaning the system can respond to many events at the same time without waiting for each to complete.

How Does Event-Driven Programming Work?

In an event-driven system, the core workflow looks like this:

1. **Event is Generated:** An event is triggered by the system, user, or external sources (e.g., a network request completes).
2. **Event Loop Monitors:** The event loop monitors the system for events, looking for changes that require attention.

3. **Event is Captured**: When an event is captured (e.g., user clicks a button), it is dispatched to the appropriate event handler.
4. **Event Handler Executes:** The event handler processes the event and triggers an action, such as updating the user interface, saving data, or sending a network request.

Each event is usually associated with a particular event handler, which is a function or method that runs when that specific event occurs. When an event is triggered, the system passes it to the relevant handler, which is responsible for responding appropriately.

Q5. Nodejs is single thread explain?

When we say Node.js is single-threaded, it means that the core of Node.js runs on a single thread. This might seem limiting at first because traditional multi-threaded architectures (such as those in Java or PHP) allow multiple tasks to be executed concurrently on different threads. However, Node.js handles concurrency and performs efficiently by relying on its non-blocking, asynchronous nature and its event-driven architecture.

Understanding Single-Threaded Architecture:

- **Single Thread:** A thread is the smallest unit of execution in a program. In Node.js, all operations, such as handling HTTP requests, file I/O, or interacting with databases, are initially handled by a single thread (the main event loop thread).
- **Non-blocking I/O:** Node.js uses non-blocking I/O operations, meaning that the single thread can initiate operations (like reading from the database or making an HTTP request), and while waiting for the result, it can continue handling other tasks or events. When the operation completes, the event loop picks up the result and triggers the appropriate callback or promise.

How Does Node.js Handle Concurrency on a Single Thread?

Node.js handles multiple tasks without blocking the event loop, using its asynchronous event-driven model. While Node.js itself is single-threaded, it delegates I/O and time-intensive tasks to background threads managed by libuv, a library used internally by Node.js to handle asynchronous operations.

Node.js handles concurrency on a single thread using an event-driven, non-blocking I/O model. Here's a breakdown of how it works:

1. **Single Threaded with an Event Loop**:
   Node.js operates on a single main thread that runs the event loop. The event loop continuously checks for tasks to be completed (like I/O operations, timers, etc.) and handles them asynchronously. This way, Node.js can manage multiple tasks without creating additional threads for each request.
2. **Asynchronous Non-Blocking I/O**:
   Most I/O operations in Node.js are non-blocking, meaning that when an I/O task (e.g., reading a file or querying a database) is initiated, it doesn't prevent the main thread from working on other tasks. Instead, Node.js offloads these tasks to the underlying OS or a thread pool provided by libuv (the C library that powers Node's asynchronous behavior). When the I/O operation is complete, a callback is queued to notify the event loop.
3. **The Event Loop and Callbacks**:
   When an asynchronous operation completes, its callback is pushed to the event loop queue, where it waits to be processed. The event loop is constantly running, handling queued callbacks and executing them as soon as the main thread is free, allowing Node.js to handle concurrent requests even though it's single-threaded.
4. **Thread Pool for Heavy Operations**:
   For operations that require more CPU power (such as file system operations or cryptography), Node.js uses a

limited thread pool provided by libuv. This thread pool can execute multiple tasks in parallel, which allows Node.js to handle several long-running tasks without blocking the main thread.

5. **Promises and Async/Await**:
   Promises and async/await provide a cleaner way to handle asynchronous operations without nesting multiple callbacks. They still leverage the event loop but allow developers to write asynchronous code that looks synchronous, making it easier to read and maintain.

   In summary, Node.js uses a single-threaded, event-driven model with non-blocking I/O operations to handle many concurrent tasks. The main thread is freed up for other tasks by offloading I/O work to the OS or a thread pool, allowing it to manage a large number of connections concurrently.

Event Loop in Action:

1. **Main Event Loop**: The single-threaded event loop continuously checks for events to process, such as new incoming HTTP requests or completion of I/O operations.
2. **Non-Blocking Asynchronous Tasks**: When Node.js needs to handle asynchronous tasks (e.g., reading a file or querying a database), it delegates these tasks to background threads (handled by libuv) while allowing the main thread to continue executing other operations.
3. **Callback Execution**: Once the background thread completes the operation (e.g., the file has been read), it sends the result back to the event loop, which then executes the callback function to process the result.

Summary:

- Single-threaded means Node.js runs its main event loop on a single thread, which is responsible for handling incoming requests and events.
- It doesn't create multiple threads for different tasks like many other programming languages do.
- However, Node.js can efficiently handle many I/O-bound tasks (such as file I/O, database queries, and network requests) by offloading them to a background thread pool managed by libuv, while the main thread remains free to handle other events.
- For CPU-intensive tasks, Node.js provides solutions like Worker Threads or Child Processes to avoid blocking the single-threaded event loop.

This single-threaded, non-blocking model makes Node.js highly efficient for I/O-heavy applications but requires careful handling of CPU-bound tasks.

Why is Node.js Single-threaded?

Node.js is single-threaded for async processing. By doing async processing on a single-thread under typical web loads, more performance and scalability can be achieved instead of the typical thread-based implementation

what is Non-blocking in nodejs

In Node.js, **non-blocking** refers to the way it handles I/O (input/output) operations, allowing the application to perform other tasks while waiting for these operations to complete. This is achieved through **asynchronous** programming, where Node.js uses callbacks, promises, or async/await to manage tasks that take time, like reading a file, making a network request, or querying a database.

In Node.js, non-blocking refers to the ability of the runtime environment to execute multiple tasks simultaneously without waiting for the completion of one task before starting the next. This is achieved through the use of asynchronous I/O operations, which allow Node.js to handle multiple requests concurrently.

How Non-Blocking Works in Node.js

In traditional, blocking (or synchronous) I/O operations, tasks are executed sequentially, meaning the code must wait for one operation to complete before moving to the next one. In contrast, with non-blocking I/O in Node.js, tasks are initiated and then control is immediately returned to the main thread, allowing it to handle other requests while the I/O operations complete in the background.

Example of Non-Blocking Behavior

Here's a basic example of a non-blocking function in Node.js:

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

// Asynchronous, non-blocking file read
fs.readFile('example.txt', 'utf8', (err, data) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log(data); // This will log the file contents when read is complete.
});

console.log("This logs immediately, not waiting for file read.");
```

In this code:

- fs.readFile starts reading the file, but control is immediately returned to the main thread.
- While readFile is running, the application continues executing the rest of the code.
- Once readFile completes, it calls the provided callback function to handle the result.

Benefits of Non-Blocking I/O

1. **High Scalability** – Node.js can handle many connections at once without waiting for each to complete, making it ideal for I/O-heavy applications like web servers and real-time apps.
2. **Efficient Resource Use** – The CPU can keep processing other requests instead of waiting idly for I/O tasks, improving application responsiveness.

Non-blocking I/O is a key reason why Node.js is well-suited for building fast, scalable network applications, especially those that require real-time data processing.

Q6. What is a Module in Node.js?

- Consider modules to be the same as JavaScript libraries.
- A set of functions you want to include in your application.
- In Node.js, **Modules** are the blocks of **encapsulated code** that communicate with an **external application on the basis of their related functionality**. Modules can be a single file or a collection of multiple files/folders. The reason programmers are heavily reliant on modules is because of their reusability as well as the ability to break down a complex piece of code into manageable chunks.

Built-in Modules

Node.js has a set of built-in modules which you can use without any further installation.

To include a module, use the require() function with the name of the module

Or we can import module in ES6 syntax.

Node.js has several built-in modules that provide a wide range of core functionalities. These modules allow you to perform file operations, networking, HTTP requests, and more without needing to install any third-party libraries. Here are some of the most commonly used built-in modules in Node.js:

1. File System (fs)

- Provides methods for working with the file system, including reading, writing, and deleting files, directories, etc.

2. Path (path)

- Used for working with file and directory paths. It provides utilities for resolving and normalizing file paths.

3. HTTP (http)

- Allows Node.js to create HTTP servers and handle HTTP requests and responses.

4. HTTPS (https)

- Similar to the http module but provides support for HTTPS requests with SSL/TLS encryption.

5. URL (url)

- Provides utilities for URL resolution and parsing.

6. Query String (querystring)

- Helps parse and format URL query strings.

7. OS (os)

- Provides information about the operating system, such as memory, CPU details, and network interfaces.

8. Process (process)

- Gives access to the current Node.js process, including environment variables, command-line arguments, and methods to exit the process.

9. Events (events)

- Provides an event-driven model with an EventEmitter class that allows you to define and listen to custom events.

10. Crypto (crypto)

- Provides cryptographic functionality such as hashing, encryption, and decryption.

11. Buffer (buffer)

- Used to handle binary data directly in Node.js, which is useful for working with files or networking protocols.

12. Stream (stream)

- Provides an interface for handling streaming data, such as reading/writing files or working with network requests.

13. Child Process (child_process)

- Allows you to spawn child processes in Node.js, enabling you to execute other programs or scripts from within a Node.js script.

14. Readline (readline)

- Provides a way to read input from a readable stream, like user input in the console.

15. Timers (timers)

- Provides functions to execute code after a set amount of time (setTimeout) or repeatedly (setInterval).

These modules come pre-installed with Node.js, so you can start using them right away by requiring them in your code. They cover a wide range of functionalities needed for basic to advanced operations in Node.js applications.

Node.js has many modules to provide the basic functionality needed for a web application. Some of them include:

Core Modules	Description
HTTP	Includes classes, methods, and events to create a Node.js HTTP server
util	Includes utility functions useful for developers
fs	Includes events, classes, and methods to deal with file I/O operations
url	Includes methods for URL parsing
query string	Includes methods to work with query string
stream	Includes methods to handle streaming data
zlib	Includes methods to compress or decompress files

Local Modules:

Unlike built-in and external modules, local modules are created locally in your Node.js application. Let's create a simple calculating module that calculates various operations. Create a calc.js file that has the following code:

```
// Filename: calc.js
exports.add = function (x, y) {
   return x + y;  };
exports.sub = function (x, y) {
   return x - y; };
exports.mult = function (x, y) {
```

```
  return x * y; };
exports.div = function (x, y) {
  return x / y; };
```

Q7. Understanding async/await in Node.js

Async/await is a native feature available in Node.js that makes it easier to manage tasks that take time, like waiting for a response from an API. In Node.js, where it's common to handle many tasks simultaneously, async/await keeps our asynchronous code organized and more readable.

async and await in Node.js are used to handle asynchronous operations more cleanly, allowing developers to write asynchronous code that looks and behaves more like synchronous code. This syntax builds on Promises, making it easier to handle multiple asynchronous operations without deep callback nesting (also known as "callback hell").

Key Concepts of async and await

1. **Async Function**:
    - Declaring a function as async means that the function will return a Promise automatically, even if you don't explicitly return a Promise.
    - You use the async keyword before a function definition:

```
async function fetchData() {
  return "Hello, world!";
}
fetchData().then(console.log);  // Prints "Hello, world!"
```

2. **Await Keyword**:

• `await` is used to pause the execution of an `async` function until a `Promise` is resolved. It can only be used inside an `async` function.
• When `await` is placed before a `Promise`, it waits until that promise is resolved and then returns the resolved value. If the promise is rejected, `await` throws the error, which can be caught with `try-catch`.

Example:

```
async function fetchData() {
  try {
    const data = await someAsyncFunction();
    console.log(data);
  } catch (error) {
    console.error(error);
  }
}
```

Example with Promises and async/await

Suppose you have a function fetchUserData that fetches user data from an API and returns a promise:

Using Promises:

```
function fetchUserData() {
  return new Promise((resolve, reject) => {
```

```
    setTimeout(() => {
      resolve("User data received");
    }, 2000);
  });
}

fetchUserData()
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

Using async/await:

```
async function getUserData() {
  try {
    const data = await fetchUserData();
    console.log(data);
  } catch (error) {
    console.error(error);
  }
}
getUserData();
```

In the async/await example, the code is cleaner and easier to read since it looks more like synchronous code. This makes it especially useful when handling multiple asynchronous operations in sequence.

Handling Multiple Async Operations with await

Using async and await also makes it easier to handle sequential and parallel execution of asynchronous tasks:

- **Sequential Execution**:

```
async function getData() {
  const data1 = await fetchData1();
  const data2 = await fetchData2();
  console.log(data1, data2);
}
```

Here, fetchData1 and fetchData2 are called sequentially, meaning fetchData2 will only start once fetchData1 has completed.

- **Parallel Execution**:

```
async function getData() {
  const [data1, data2] = await Promise.all([fetchData1(), fetchData2()]);
  console.log(data1, data2);
}
```

By using Promise.all, fetchData1 and fetchData2 are started simultaneously, which is more efficient when the operations are independent of each other.

Error Handling in async/await

Errors in asynchronous code can be handled with try-catch blocks:

```
async function getData() {
  try {
    const data = await fetchData();
```

```
    console.log(data);
  } catch (error) {
    console.error("Error:", error);
  }
}
```

If fetchData rejects, the error will be caught by the catch block.

Benefits of async/await

- **Readable Code**: Asynchronous code looks more like synchronous code, making it easier to read and understand.
- **Error Handling**: Errors can be managed using try-catch, which feels more natural than handling errors in .catch for Promises.
- **Structured Flow**: Sequential or parallel execution of async tasks can be controlled easily.

Limitations of async/await

- **Blocking Nature**: Using await in a loop or with heavy operations can lead to performance bottlenecks since each await pauses the function until the promise is resolved.
- **Only Works in Async Functions**: await can only be used inside functions marked with async.

In summary, async and await provide a cleaner and more manageable way to handle asynchronous code in Node.js, reducing callback nesting and making error handling more straightforward.

❖ **How to declare an async function**

We know that async/await helps us handle asynchronous operations, but how do we use it?

It's as easy as declaring a function with the async keyword. All async functions return a promise, automatically encapsulating non-promise values. Functions can be made async by using the keyword async before the function declaration:

```
async function bakeCookies() {
  // Simulating baking time
  await new Promise((resolve) => setTimeout(resolve, 2000));
}
```

❖ **What is await?**

Await is a keyword paired with async functions that causes the JavaScript interpreter to pause until the following asynchronous operation is run. You can also assign the value to a variable to use later. It's important to remember that the keyword await can only be placed in the bodies of async functions.

```
async function serveCookies() {
  const cookies = await bakeCookies();
  console.log("The cookies are now ready to serve!");
}


serveCookies()
```

If we assume that bakeCookies() in the example above is an asynchronous operation that returns a promise, the await keyword will halt the function serveCookies() until bakeCookies() has returned a value.

❖ Async and await

async and await are JavaScript features that allow you to handle asynchronous operations more cleanly

and concisely. They help simplify working with promises by making asynchronous code look and behave

more like synchronous code, which is easier to read and maintain.

❖ Why Use async and await?

When working with tasks like fetching data from an API, reading a file, or performing any operation that takes time (I/O operations), the JavaScript engine doesn't wait for the task to finish.

It continues running other code. To handle this asynchronous behavior, JavaScript originally used callbacks and later promises. However, these can lead to complex and hard-to-read code, especially if you have many asynchronous operations to chain together.

async and await were introduced to solve these problems by allowing asynchronous code to be written in a more synchronous, readable manner.

Use async/await in Node.js when you want to write asynchronous code that is more readable, sequential, and better at error handling.

This approach reduces the cognitive load for developers, making it easier to understand, read, and debug the code. By maintaining a top-to-bottom flow and utilizing try/catch for error handling, async/await simplifies the learning curve for those who are new to asynchronous programming.

❖ How async and await Work:

async: The async keyword is placed before a function to indicate that the function will return a promise. Even if the function seems to return a value immediately, it is wrapped in a promise.

await: The await keyword is used inside an async function to pause the execution of the function until a promise is resolved or rejected. It allows you to "wait" for a promise to be resolved and get its result as if the code were synchronous**.**

❖ If await pause the execution until a promise is return then how it execute the code asynchronously what does that mean.

The confusion around async/await often arises because it appears to make asynchronous code behave synchronously. Let's break it down:

What await Does

When you use await, it pauses the execution of the **current function** until the Promise it is waiting for is resolved or rejected. However, this does **not block the entire program**—only the function that contains the await is paused. Other parts of your program or other asynchronous tasks continue to execute.

Asynchronous Execution with await

The key to understanding await is realizing that it **does not make the code synchronous**. Instead, it allows you to write asynchronous code that looks synchronous for better readability.

Here's what happens when await is used:

1. **Encountering await:**
    - When the JavaScript engine encounters await, it pauses the execution of the containing async function and hands control back to the event loop.
    - Other tasks in the event loop (like handling user input, processing other async functions, or running setTimeout callbacks) continue.
2. **Resolving the Promise:**
    - Once the awaited Promise resolves or rejects, the JavaScript engine resumes execution of the paused async function where it left off.
3. **Execution Resumes:**
    - After resuming, the async function continues its execution as if the await line had returned the resolved value.

Example

```
async function fetchData() {
  console.log("Start fetching...");

  // Simulates an asynchronous task
  const data = await new Promise(resolve => {
    setTimeout(() => resolve("Fetched Data"), 2000);
  });

  console.log("Data received:", data);
}

console.log("Before fetchData");
fetchData();
console.log("After fetchData");
```

Step-by-Step Execution

1. **Before fetchData:**
    - console.log("Before fetchData") is executed immediately.
2. **Calling fetchData:**
    - The fetchData function starts executing.
    - It logs "Start fetching...".
    - Then it encounters await.
3. **At await:**
    - The promise starts running in the background.
    - Execution of the fetchData function is paused.

- Control returns to the event loop, and "After fetchData" is logged.

4. **Promise Resolves:**
    - After 2 seconds, the promise resolves.
    - The paused fetchData function resumes and logs "Data received: Fetched Data".

Why It's Still Asynchronous

- **No Blocking:** Even though await pauses the fetchData function, it doesn't block the entire program. Other tasks in the event loop continue to run.
- **Non-Blocking Nature of JavaScript:** JavaScript can continue processing other async tasks or code outside the fetchData function while waiting for the promise.

Comparison with Synchronous Code

If the same logic were synchronous, the entire program would halt at await until the promise resolved.

```
function fetchDataSync() {
  const data = longRunningSyncOperation(); // Blocks the program
  console.log("Data received:", data);
}

console.log("Before fetchDataSync");
fetchDataSync(); // Nothing else runs until this is done
console.log("After fetchDataSync"); // This is blocked until the function finishes
```

Summary

- await **pauses only the containing async function**, not the entire program.
- It allows other asynchronous tasks or event-loop operations to proceed while waiting for a promise to resolve.
- This combination makes async/await both asynchronous and non-blocking while maintaining readability that feels synchronous.

This design provides the benefits of asynchronous programming without the complexity of nested callbacks or explicitly handling promises with .then().

❖ When to Use async and await:

When handling asynchronous operations: If your code is performing tasks like fetching data, reading files, interacting with databases, or any other operation that involves waiting for something to complete.

When you want readable, maintainable code: async and await make asynchronous code look much more like regular synchronous code, improving readability.

When chaining promises: Instead of chaining .then() calls, await simplifies the process by flattening the code into a straightforward, step-by-step flow.

Key Benefits:

Improved readability: Asynchronous code can be written in a sequential style, making it easier to understand**.**

Better error handling: async/await uses try-catch blocks for error handling, making it simpler to manage errors compared to using .catch() with promises.

Easier debugging: Since async/await code looks more like synchronous code, it's easier to trace and debug**.**

Important Points:

await can only be used inside an async function: You cannot use await outside of an async function.

Error handling: You should always wrap await in a try-catch block to handle any errors from the promise.

Parallelism: If you want to perform multiple asynchronous tasks in parallel, avoid using await sequentially.

Instead, you can use Promise.all:

```
async function fetchMultiple() {
  const [data1, data2] = await Promise.all([
    fetch('https://api.example.com/data1'),
    fetch('https://api.example.com/data2')
  ]);
}
```

Explain fetch in Nodejs??

In **Node.js**, fetch is a method used to make HTTP requests to external APIs or servers. It allows your application to interact with resources over the internet, such as fetching data from a REST API, submitting data to a server, or performing CRUD operations.

The fetch API was originally introduced in browsers for client-side JavaScript. In Node.js, it became natively available starting from **Node.js v18**, as part of the global scope. For earlier versions, developers commonly used libraries like node-fetch or axios.

Key Features of fetch

1. **Promise-Based:** fetch returns a Promise, making it easier to handle asynchronous requests with .then() and .catch(), or using async/await.
2. **Stream Handling:** The response is provided as a stream, allowing efficient handling of large payloads.
3. **Customizable Requests:** You can specify HTTP methods (GET, POST, PUT, DELETE, etc.), headers, and body content.

Syntax

```
fetch(url, options)
  .then(response => {
    // Handle the response
  })
  .catch(error => {
    // Handle errors
  });
```

Parameters

- **url**: The URL of the resource you want to fetch.

- **options**: An optional object containing request configuration, such as method, headers, body, etc.

Example Usage in Node.js

1. Basic GET Request

```
const fetch = require('node-fetch'); // For Node.js versions < 18

fetch('https://jsonplaceholder.typicode.com/posts/1')
  .then(response => {
    if (!response.ok) {
      throw new Error(`HTTP error! Status: ${response.status}`);
    }
    return response.json(); // Parse JSON response
  })
  .then(data => {
    console.log(data); // Log the fetched data
  })
  .catch(error => {
    console.error('Error:', error.message);
  });
```

2. POST Request with Body and Headers

```
fetch('https://jsonplaceholder.typicode.com/posts', {
  method: 'POST', // HTTP method
  headers: {
    'Content-Type': 'application/json', // Specify content type
  },
  body: JSON.stringify({
    title: 'foo',
    body: 'bar',
    userId: 1,
  }),
})
  .then(response => response.json())
  .then(data => {
    console.log('Created Post:', data);
  })
  .catch(error => {
    console.error('Error:', error.message);
  });
```

JavaScript JSON stringify() Method

The JSON.stringify() method in JavaScript is used to convert JavaScript objects into a JSON string. This method takes a JavaScript object as input and returns a JSON-formatted string representing that object.

```
console.log(JSON.stringify({ x: 5, y: 6 }));
// Expected output: '{"x":5,"y":6}'

console.log( JSON.stringify([new Number(3), new String('false'), new Boolean(false)]),);
// Expected output: '[3,"false",false]'
```

```
console.log(JSON.stringify( { x: [10, undefined, function () {}, Symbol('')]} ));
// Expected output: '{"x":[10,null,null,null]}'

console.log(JSON.stringify(new Date(2006, 0, 2, 15, 4, 5)));
// Expected output: '"2006-01-02T15:04:05.000Z"'
```

.json() in Node.js

The .json() method in Node.js is commonly used in frameworks like Express.js to handle JSON data.

Sending JSON Responses

In Express.js, res.json() is used to send a JSON response to the client. It automatically sets the Content-Type header to application/json.

3. Using async/await

```
const fetch = require('node-fetch'); // Not required in Node.js >= 18

const fetchData = async () => {
  try {
    const response = await fetch('https://jsonplaceholder.typicode.com/posts/1');
    if (!response.ok) {
      throw new Error(`HTTP error! Status: ${response.status}`);
    }
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error('Error:', error.message);
  }
};

fetchData();
```

Error Handling

fetch does **not reject the Promise** for HTTP errors like 404 or 500. You must check the response.ok property or the status code manually.

```
fetch('https://example.com/data')
  .then(response => {
    if (!response.ok) {
      throw new Error(`HTTP error! Status: ${response.status}`);
    }
    return response.json();
  })
  .then(data => console.log(data))
  .catch(error => console.error('Error:', error.message));
```

The fetch() function will reject the promise on some errors, but not if the server responds with an error status like 404: so we also check the response status and throw if it is not OK.

Otherwise, we fetch the response body content as JSON by calling the json() method of Response, and log one of its values. Note that like fetch() itself, json() is asynchronous, as are all the other methods to access the response body content.

Setting cors:

```
const response = await fetch("https://example.org/post", {
  mode: "cors",
  // ...
});
```

Setting the method

By default, fetch() makes a GET request, but you can use the method option to use a different request method:

```
const response = await fetch("https://example.org/post", {
  method: "POST",
  // ...
});
```

If the mode option is set to no-cors, then method must be one of GET, POST or HEAD.

Setting a body

The request body is the payload of the request: it's the thing the client is sending to the server. You cannot include a body with GET requests, but it's useful for requests that send content to the server, such as POST or PUT requests. For example, if you want to upload a file to the server, you might make a POST request and include the file as the request body.

To set a request body, pass it as the body option:

```
const response = await fetch("https://example.org/post", {
  body: JSON.stringify({ username: "example" }),
  // ...
});
```

You can supply the body as an instance of any of the following types:

- a string
- ArrayBuffer
- TypedArray
- DataView
- Blob
- File
- URLSearchParams
- FormData
- ReadableStream

Note that just like response bodies, request bodies are streams, and making the request reads the stream, so if a request contains a body, you can't make it twice:

```
const request = new Request("https://example.org/post", {
  method: "POST",
```

```
  body: JSON.stringify({ username: "example" }),
});

const response1 = await fetch(request);
console.log(response1.status);

// Will throw: "Body has already been consumed."
const response2 = await fetch(request);
console.log(response2.status);
```

Instead, you would need to create a clone of the request before sending it:

```
const request1 = new Request("https://example.org/post", {
  method: "POST",
  body: JSON.stringify({ username: "example" }),
});

const request2 = request1.clone();

const response1 = await fetch(request1);
console.log(response1.status);

const response2 = await fetch(request2);
console.log(response2.status);
```

See Locked and disturbed streams for more information.

Setting headers

Request headers give the server information about the request: for example, the Content-Type header tells the server the format of the request's body.

To set request headers, assign them to the headers option.

You can pass an object literal here containing header-name: header-value properties:

```
const response = await fetch("https://example.org/post", {
  headers: {
    "Content-Type": "application/json",
  },
  // ...
});
```

Alternatively, you can construct a Headers object, add headers to that object using Headers.append(), then assign the Headers object to the headers option:

```
const myHeaders = new Headers();
myHeaders.append("Content-Type", "application/json");

const response = await fetch("https://example.org/post", {
  headers: myHeaders,
  // ...
});
```

Many headers are set automatically by the browser and can't be set by a script: these are called Forbidden header names. If the mode option is set to no-cors, then the set of permitted headers is further restricted.

Considerations

1. **Streaming:** fetch streams the response. You need to explicitly parse it (e.g., response.json() for JSON data).
2. **Timeouts:** fetch does not have built-in timeout support. You may need additional libraries or logic to handle timeouts.
3. **Cross-Origin Requests:** fetch in Node.js is not restricted by CORS policies as it is on the browser.

By using fetch in Node.js, you can easily integrate external APIs into your applications with a clean and modern API

Q8. What is a callback?

Node.js callbacks are a special type of function passed as an argument to another function.

They're called when the function that contains the callback as an argument completes its execution, and allows the code in the callback to run in the meantime.

A Callback in Node.js is an asynchronous equivalent for a function. It is a special type of function passed as an argument to another function. Node.js makes heavy use of callbacks. Callbacks help us make asynchronous calls. All the APIs of Node are written in such a way that they support callbacks.

Programming instructions are executed synchronously by default. If one of the instructions in a program is expected to perform a lengthy process, the main thread of execution gets blocked. The subsequent instructions can be executed only after the current I/O is complete. This is where callbacks come in to the picture.

The callback is called when the function that contains the callback as an argument completes its execution, and allows the code in the callback to run in the meantime. This makes Node.js highly scalable, as it can process a high number of requests without waiting for any function to return results.

- Callbacks are nothing but functions that take some time to produce a result.
- Usually, these async callbacks (async short for asynchronous) are used for accessing values from databases, downloading images, reading files, etc.
- As these take time to finish, we can neither proceed to the next line because it might throw an error saying unavailable nor we can pause our program.
- So we need to store the result and call back when it is complete.

Example of a Callback:

```
app.get('/', function(){
     function1(arg1, function(){
        ...
    })
});
```

The problem with this kind of code is that this kind of situations can cause a lot of trouble and the code can get messy when there are several functions. This situation is called what is commonly known as a **callback hell**. So, to find a way out, the idea of **Promises and function chaining** was introduced.

Q9. What is callback hell?

Callback hell (also known as "pyramid of doom") is a common problem in asynchronous programming, particularly when using callbacks. It occurs when multiple asynchronous operations need to be performed sequentially, and each step depends on the result of the previous step. This results in a deeply nested structure of callbacks, making the code difficult to read, maintain, and debug.

This is a big issue caused by coding with complex nested callbacks. Here, each and every callback takes an argument that is a result of the previous callbacks. In this manner, the code structure looks like a pyramid, making it difficult to read and maintain. Also, if there is an error in one function, then all other functions get affected.

Callback hell occurs in JavaScript when handling multiple asynchronous operations using nested callbacks, resulting in complicated, hard-to-read code.

Why Callback Hell Happens

In JavaScript, asynchronous operations (like reading a file, making a network request, or querying a database) are often handled with callbacks. When you have a series of asynchronous tasks that must be executed in order, each callback function calls another asynchronous operation, leading to code that is deeply indented and complex.

Key Pointers

- Callback hell arises from complex, nested callbacks.
- It creates a pyramid structure that is hard to read and maintain.
- Errors in one function can affect others.
- An example involves nested functions handling results from previous functions.
- JavaScript offers promises to escape callback hell, using the .then method for orderly chaining.
- Promises use the .catch method to handle exceptions.
- async and await keywords, and functions like setTimeout, improve readability and maintainability.

Q10. What is a Promise?

A **Promise** in JavaScript is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value. Promises provide a more manageable way to handle asynchronous operations compared to traditional callback functions, especially when chaining multiple operations. Allowing you to manage operations that might take time to complete, like reading files, making HTTP requests, or interacting with databases.

A promise in JavaScript is like a container for a future value. It is a way of saying, “I don’t have this value right now, but I will have it later.” Imagine you order a book online. You don’t get the book right away, but the store promises to send it to you. While you wait, you can do other things, and when the book arrives, you can read it. In the same way, a promise lets you keep working with your code while waiting for something else to finish, like loading data from a server. When the data is ready, the promise will deliver it.

A promise is basically an advancement of callbacks in Node. In other words, a promise is a JavaScript object which is used to handle all the asynchronous data operations. While developing an application you may encounter that you are using a lot of nested callback functions.

Now imagine if you need to perform multiple nested operations like this. That would make your code messy and very complex. In Node.js world, this problem is called **"Callback Hell"**. To resolve this issue we need to get rid of the callback functions whilst nesting. This is where Promises come into the picture. A Promise in Node means an action which will either be completed or rejected. In case of completion, the promise is kept and otherwise, the promise is broken. So as the word suggests either the promise is kept or it is broken. And unlike callbacks, promises can be chained.

How Does a Promise Work?

A promise can be in one of three states:

- **Pending**: The promise is waiting for something to finish. For example, waiting for data to load from a website.
- **Fulfilled**: The promise has been completed successfully. The data you were waiting for is now available.
- **Rejected**: The promise has failed. Maybe there was a problem, like the server not responding.

When you create a promise, you write some code that will eventually tell the promise whether it was successful (fulfilled) or not (rejected).

Syntax:

```
let promise = new Promise(function(resolve, reject){
    //do something
});
```

Parameters

- The promise constructor takes only one argument which is a callback function
- The callback function takes two arguments, *resolve* and *reject*
    - Perform operations inside the callback function and if everything went well then call resolve.
    - If desired operations do not go well then call reject.

Advantages of using promises instead of callbacks?

- The control flow of asynchronous logic is more specified and structured.
- The coupling is low.
- We've built-in error handling.
- Improved readability.

Q11. How to handle large file in nodejs

Handling large files in Node.js can be challenging because reading an entire file into memory at once can lead to memory exhaustion(Consumption), especially if the file is several gigabytes in size. Node.js provides efficient ways to deal with large files by using **streams** and other techniques that allow you to process files in smaller chunks rather than loading the entire file into memory at once.

Key Methods to Handle Large Files in Node.js

1. **Using Streams**: Streams in Node.js allow you to process data incrementally, reading or writing small chunks of data at a time, rather than loading everything at once. Node.js has built-in support for streams, which makes it a great tool for handling large files.
2. **Using fs.createReadStream() for Reading**: This method is part of the built-in fs (file system) module and allows you to read files in chunks (also known as **streaming**). Instead of reading the whole file into memory, it reads the file in chunks and processes it piece by piece.
3. **Using fs.createWriteStream() for Writing**: This method allows you to write data in smaller chunks to avoid loading large amounts of data into memory.

Example: Using Streams to Read and Write a Large File

1. Reading a Large File Using Streams

The most efficient way to read a large file in Node.js is by using the fs.createReadStream() function.

```
const fs = require('fs');
// Create a readable stream from the large file
const readStream = fs.createReadStream('largefile.txt', { encoding: 'utf8' });
// Handle data event - data is received in chunks
readStream.on('data', (chunk) => {
  console.log('Received a chunk:', chunk);
});
// Handle end event - called when the stream is done reading
readStream.on('end', () => {
  console.log('Finished reading file.');
});
// Handle error event - called in case of errors during reading
readStream.on('error', (err) => {
  console.error('An error occurred:', err);
});
```

- **Explanation**:
    - The fs.createReadStream() method reads the file in chunks, and every time a chunk of data is ready, it emits a data event.
    - When the entire file has been read, the end event is triggered.
    - This approach ensures that the memory usage remains low because only a small portion of the file is kept in memory at any given time.

2. Writing to a Large File Using Streams

To write to a large file efficiently, you can use fs.createWriteStream(), which allows you to write data in chunks.

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

// Create a writable stream to a large file
const writeStream = fs.createWriteStream('outputfile.txt');

// Write some data to the file
writeStream.write('This is the first chunk of data.\n');
writeStream.write('This is the second chunk of data.\n');

// End the stream after writing all the data
writeStream.end('Finished writing to the file.\n');
```

```
// Handle finish event - called when the stream is done writing
writeStream.on('finish', () => {
  console.log('Finished writing to the file.');
});

// Handle error event - called in case of errors during writing
writeStream.on('error', (err) => {
  console.error('An error occurred:', err);
});
```

- **Explanation**:
    - fs.createWriteStream() allows you to write data incrementally. When you call write(), it writes chunks of data to the file.
    - You call end() when you're finished writing data.
    - Using streams for writing avoids loading the entire data into memory at once.

3. Piping Read and Write Streams

Node.js streams allow you to **pipe** data from one stream to another, making it easy to read a large file and write it to another file without loading the entire file into memory.

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

// Create readable and writable streams
const readStream = fs.createReadStream('largefile.txt');
const writeStream = fs.createWriteStream('outputfile.txt');

// Pipe the readable stream into the writable stream
readStream.pipe(writeStream);

// Handle finish event
writeStream.on('finish', () => {
  console.log('Finished writing the file.');
});

// Handle error event
readStream.on('error', (err) => console.error('Read error:', err));
writeStream.on('error', (err) => console.error('Write error:', err));
```

- **Explanation**:
    - The pipe() method transfers data from the read stream to the write stream efficiently. It reads chunks from the file and writes them to another file, ensuring low memory usage.

Other Techniques for Handling Large Files

4. Using stream Module for Advanced Control

Node.js also provides more advanced control over streams via the stream module. You can create custom streams for more specific use cases, such as processing the file line by line, transforming data as it passes through the stream, etc.

5. Using Compression (e.g., zlib)

When working with large files, you might also want to compress or decompress files as part of your process to reduce disk usage and increase speed.

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

// Create readable and writable streams
const readStream = fs.createReadStream('largefile.txt');
const writeStream = fs.createWriteStream('largefile.txt.gz');

// Create a Gzip stream to compress the data
const gzip = zlib.createGzip();

// Pipe the read stream into the Gzip stream, then pipe it to the write stream
readStream.pipe(gzip).pipe(writeStream);

// Handle finish event
writeStream.on('finish', () => {
  console.log('File successfully compressed');
});
```

This example compresses a large file while reading and writing it using streams.

6. Splitting Large Files into Chunks

If the file is too large to handle even with streams (or if you want to distribute the load), you can split the file into smaller chunks and process each chunk individually.

For example, you can split a large text file into smaller parts, process each part, and then merge them later.

Handling File Uploads of Large Files

When dealing with large file uploads (e.g., in a web application), you can use libraries like **Multer** or **Busboy** that handle file uploads in chunks without overloading memory.

npm install multer

Example of using Multer to handle file uploads:

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

// Configure multer storage
const storage = multer.diskStorage({
  destination: function (req, file, cb) {
    cb(null, 'uploads/');
  },
  filename: function (req, file, cb) {
    cb(null, file.originalname);
  }
});

const upload = multer({ storage: storage });

// Route to handle file upload
app.post('/upload', upload.single('file'), (req, res) => {
  console.log('File uploaded successfully');
  res.send('File uploaded successfully');===]
});

app.listen(3000, () => {
```

```
  console.log('Server is running on port 3000');
});
```

Here, Multer streams the file directly to disk, ensuring that large file uploads are handled efficiently without being stored in memory.

Summary of Best Practices for Handling Large Files:

1. **Use Streams**: Always use streams (fs.createReadStream() and fs.createWriteStream()) to read and write large files in chunks, minimizing memory usage.
2. **Use Piping**: Pipe streams when reading from and writing to files to ensure smooth, efficient handling of large files.
3. **Compression**: Use libraries like zlib to compress or decompress large files, which can save disk space and speed up file handling.
4. **File Uploads**: Use libraries like Multer to handle file uploads in chunks to avoid overloading the server's memory.
5. **Error Handling**: Always implement error handling to catch issues with reading or writing large files.

These techniques will help you manage large files efficiently in Node.js applications while avoiding memory bottlenecks.

Q12. Mention the benefits of using Node.js.

Below are the main benefits of using Node.js:

- The APIs in Node.js are asynchronous which means that they do not cause blockers. This happens due to the Node.js server not waiting for the return of data once an API has been called. It moves on to the next API. Instead, there is a notification mechanism that gets the response from the prior API call. Thus, we can also say that Node.js is event-driven.
- Node.js is powered by the Google Chrome V8 JavaScript Engine. This makes code execution in the Node.js library super-fast.
- Node.js is based on a single-thread model with event looping. The event mechanism in the Node.js library makes it highly scalable because it prevents blockers on the server as compared to other servers that can only handle limited requests. In Node.js, the single thread program can service copious requests compared to something like Apache HTTP Server.
- Because of the event mechanism in Node.js, there is no buffering. Instead, the data output occurs in chunks.

Benefits of using Node.js

1. **High Performance with V8 Engine**
2. **Event-Driven, Non-Blocking I/O**
3. **Single Programming Language (JavaScript)**
4. **Scalability for Microservices and Distributed Systems**
5. **Large and Active Community**
6. **Real-Time Applications**
7. **Cost-Effective and Efficient for Development Teams**
8. **JSON Support for API Development**
9. **Cross-Platform Development**
10. **Strong Corporate Backing and Wide Adoption**
11. **Built-in Support for Package Management with NPM**
12. **Improved Developer Productivity**
13. **Ease of Deployment and DevOps Support**

Q13. Explain global installation of dependencies.

/npm directory stores the globally installed dependencies. While these dependencies can be used in the Command Line Interface or CLI function of all Node.js libraries, they cannot be directly imported in the Node application with the require() command. If you want to globally install a dependency, you can use the -g flag.

When you install dependencies (libraries, frameworks, or tools) in Node.js, you can do so either **locally** or **globally**. Understanding the difference between these types of installations is crucial for managing packages effectively.

Q14. What is Global Installation of Dependencies?

A **global installation** of a dependency or package means that the package is installed system-wide, making it available for use from any location on your machine. In contrast to local installations (which are installed in the node_modules folder within a specific project), globally installed packages can be used in any project or directly from the command line as a command-line interface (CLI) tool.

Global installation is mostly used for tools or utilities that you need to access across different projects or directly through the terminal.

How to Globally Install a Package

To globally install a package using Node's package manager (npm or yarn), you use the -g or --global flag with the installation command.

npm install -g <package-name>

npm install -g nodemon

Local Installation vs Global Installation

Aspect	**Local Installation**	**Global Installation**
Command	npm install <package-name>	npm install -g <package-name>
Installation Scope	Only available within the specific project	Available system-wide
Installed Location	node_modules in the project directory	A system-wide directory (e.g., /usr/local/lib/node_modules on Linux)
Usage	Only accessible within the project	Can be used in any project or globally from the terminal
Typical Use Case	Project-specific dependencies (e.g., libraries)	Tools/CLI utilities (e.g., nodemon, typescript)

Q15. Explain REPL in Node.js.

REPL stands for Read Eval Print Loop. REPL performs tasks related to reading, executing, printing, and looping. The server contains several ad-hoc Java statements that REPL can help execute. With REPL, the

JavaScript can enter the shell post directly and reliably perform tasks like debugging, testing, and experimenting.

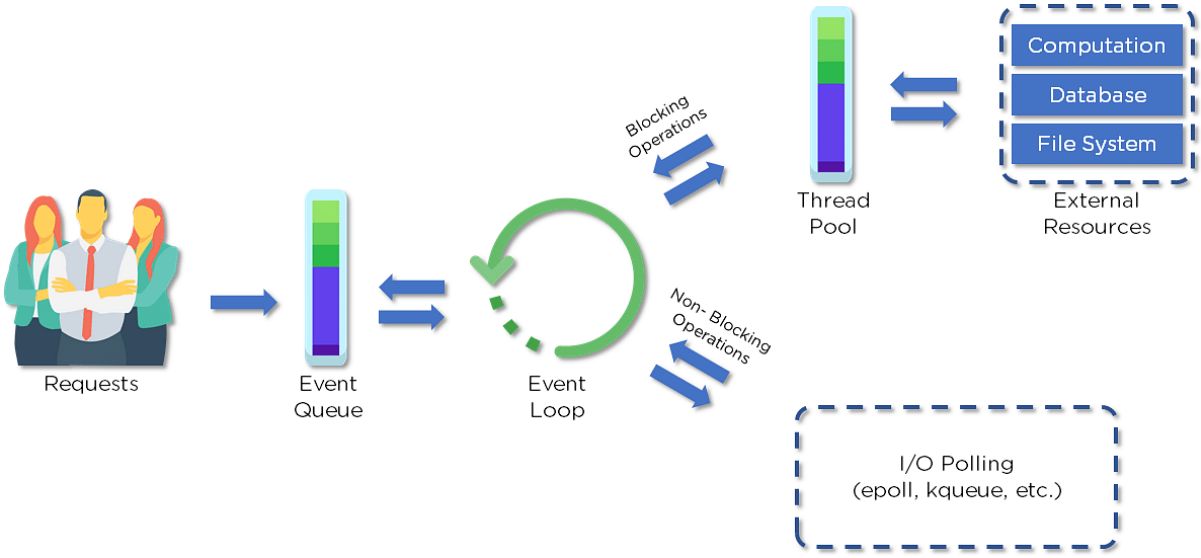
Q16. Why use Node.js?

Node.js makes building scalable network programs easy. Some of its advantages include:

- It is generally fast
- It rarely blocks
- It offers a unified programming language and data type
- Everything is asynchronous
- It yields great concurrency

Q17. How does Node.js work?

A web server using Node.js typically has a workflow that is quite similar to the diagram illustrated below. Let's explore this flow of operations in detail.



- Clients send requests to the webserver to interact with the web application. Requests can be non-blocking or blocking:
- Querying for data
- Deleting data
- Updating the data
- Node.js retrieves the incoming requests and adds those to the Event Queue.
- The requests are then passed one-by-one through the Event Loop. It checks if the requests are simple enough not to require any external resources.
- The Event Loop processes simple requests (non-blocking operations), such as I/O Polling, and returns the responses to the corresponding clients.

A single thread from the Thread Pool is assigned to a single complex request. This thread is responsible for completing a particular blocking request by accessing external resources, such as computation, database, file system, etc.

Once the task is carried out completely, the response is sent to the Event Loop that sends that response back to the client.

Q18. What makes Node.js different?

It is different from other JavaScript environments because it is asynchronous and event-driven.

Q19. How would you define the term I/O?

- The term I/O is used to describe any program, operation, or device that transfers data to or from a medium and to or from another medium
- Every transfer is an output from one medium and an input into another. The medium can be a physical device, network, or files within a system
- I/O stands for input/output, which helps write and read files and network operations.

Q20. What can we build with Node.js?

Node.js developers can build various applications, including web applications, chat applications, real-time applications, streaming applications, APIs, and desktop applications, etc.

Q21. Does Node.js use JavaScript?

Node.js is based on JavaScript and it uses the V8 engine developed by Google. It is used for building server-side applications.

Q22. What is NPM?

NPM stands for Node Package Manager, responsible for managing all the packages and modules for Node.js.

Npm stands for the Node package manager. It is used for installing, updating, and uninstalling packages in your application. It helps to manage dependencies in Node.js applications.

Node Package Manager provides two main functionalities:

- Provides online repositories for node.js packages/modules, which are searchable on search.nodejs.org
- Provides command-line utility to install Node.js packages and also manages Node.js versions and dependencies

Q23. what is ORM in nodejs

In Node.js, an ORM (Object-Relational Mapping) is a tool that simplifies database operations by allowing developers to interact with a relational database using an object-oriented paradigm instead of writing raw SQL queries. ORMs map database tables to JavaScript objects, enabling developers to perform database operations through the use of methods and properties on these objects rather than directly interacting with SQL syntax.

An **ORM (Object-Relational Mapping)** is a library or tool in Node.js that allows developers to interact with relational databases (like MySQL, PostgreSQL, or SQLite) in an object-oriented way, rather than writing raw SQL queries.

ORMs provide:

1. **Abstraction:** Tables map to classes, rows to objects, and columns to object properties.
2. **Convenience:** CRUD operations (Create, Read, Update, Delete) and complex queries are simplified.
3. **Portability:** The same code can often work across multiple database types with minimal changes.

Popular ORMs in Node.js

- **Sequelize**: A widely used ORM for Node.js that supports MySQL, PostgreSQL, SQLite, and MariaDB.
- **TypeORM**: An ORM built with TypeScript and JavaScript, also supporting MySQL, PostgreSQL, SQLite, and more.
- **Bookshelf.js**: A lightweight ORM built on top of Knex.js.

How to Use an ORM (Example with Sequelize)

Setup Example with Sequelize

1. **Install Dependencies**

   bash

   Copy code

   npm install sequelize mysql2

   - sequelize: The ORM library.
   - mysql2: A driver for connecting to MySQL.

---

2. **Database Connection** Create a db.js file for connecting to the database.

   javascript

   Copy code

```
const { Sequelize } = require('sequelize');

const sequelize = new Sequelize('database_name', 'username', 'password', {
  host: 'localhost',
  dialect: 'mysql', // Use 'postgres', 'sqlite', etc., for other databases.
});

module.exports = sequelize;
```

---

3. **Define a Model** Create a models/User.js file.

   javascript

   Copy code

```
const { DataTypes } = require('sequelize');
const sequelize = require('../db'); // Import the connection

const User = sequelize.define('User', {
  // Define the columns
```

```
  id: {
    type: DataTypes.INTEGER,
    primaryKey: true,
    autoIncrement: true,
  },
  name: {
    type: DataTypes.STRING,
    allowNull: false,
  },
  email: {
    type: DataTypes.STRING,
    unique: true,
    allowNull: false,
  },
});

module.exports = User;
```

---

4. **Perform Operations** Use the defined model to interact with the database.

```
const sequelize = require('./db'); // Import the database connection
const User = require('./models/User'); // Import the User model

(async () => {
  try {
    // Sync the database (create the table if it doesn't exist)
    await sequelize.sync();

    // Create a new user
    const newUser = await User.create({
      name: 'John Doe',
      email: 'john.doe@example.com',
    });
    console.log('User created:', newUser.toJSON());
```

```
    // Read users
    const users = await User.findAll();
    console.log('All Users:', users.map(user => user.toJSON()));

    // Update a user
    const updatedUser = await User.update(
      { name: 'Jane Doe' },
      { where: { email: 'john.doe@example.com' } }
    );
    console.log('User updated:', updatedUser);

    // Delete a user
    await User.destroy({ where: { email: 'john.doe@example.com' } });
    console.log('User deleted');
  } catch (error) {
    console.error('Error:', error);
  }
})();
```

Code Walkthrough

1. **Database Connection (db.js)**:
   - Establishes a connection to the database using Sequelize.
2. **Model Definition (models/User.js)**:
   - Maps a User table with columns id, name, and email to the User model.
3. **CRUD Operations**:
   - **Create**: User.create() inserts a new row into the table.
   - **Read**: User.findAll() retrieves all rows from the table.
   - **Update**: User.update() modifies data based on conditions.
   - **Delete**: User.destroy() deletes rows matching the condition.

Advantages of Using ORM

1. **Readability**:
   - Simplifies database operations, making code more intuitive and maintainable.

2. **Abstraction**:
    - Developers can focus on the business logic without worrying about SQL syntax.
3. **Cross-Database Compatibility**:
    - Makes it easier to switch between database systems.
4. **Built-in Features**:
    - Validations, associations (relationships like one-to-many), and migrations are often built into ORMs.

Considerations When Using ORMs

1. **Overhead**:
    - ORMs can introduce performance overhead compared to raw SQL.
2. **Complex Queries**:
    - Writing very complex queries might be easier with raw SQL.
3. **Learning Curve**:
    - Learning to use an ORM effectively requires understanding its features and limitations.

This example shows how ORMs like Sequelize simplify working with relational databases in Node.js while providing a high level of abstraction and functionality.

Q24. What are the two data types categories in Node.js?

Node.js supports two categories of data type - primitive and non-primitive.

Same as javaScript data type.

Q25. Name types of API functions supported by Node.js.

Asynchronous non-blocking and synchronous blocking are the two different types of API functions that Node.js supports.

Q26. Why does Google use V8 for Node.js?

Google uses V8 for Node.js because it is faster and more efficient. It compiles the JavaScript code directly into machine code.

Q27. What is the purpose of the module .Exports?

In Node.js, a module encapsulates all related codes into a single unit of code that can be parsed by moving all relevant functions into a single file. You may export a module with the module and export the function, which lets it be imported into another file with a needed keyword.

Q28. Why is Node.js preferred over other backend technologies like Java and PHP?

Some of the reasons why Node.js is preferred include:

- Node.js is very fast
- Node Package Manager has over 50,000 bundles available at the developer's disposal

- Perfect for data-intensive, real-time web applications, as Node.js never waits for an API to return data
- Better synchronization of code between server and client due to same code base
- Easy for web developers to start using Node.js in their projects as it is a JavaScript library

Q29. For what type of applications is Node.js not compatible?

Node.js is not compatible with CPU-intensive applications.

CPU-intensive applications are software programs or processes that require a high amount of CPU (Central Processing Unit) power to execute efficiently. These applications involve complex calculations or data processing tasks that rely heavily on the CPU's speed and ability to handle multiple instructions simultaneously.

Examples of CPU-intensive applications include:

1. **Video Rendering and Editing Software**:
2. **Scientific Simulations and Calculations**
3. **Machine Learning and AI**:
4. **3D Gaming and Graphics Applications**:
5. **Financial Modeling and Trading**:

Q30. What is Node red?

Node red is a visual programming tool for Node.js that is used to wire hardware devices and online services as part of IoT applications.

Q31. How is operational error different from programming error?

An operational error occurs naturally and is part of the application flow, while programming errors are referred to as bugs that are caused by developers.

Q32. What is unit testing?

Unit testing in Node.js is a process of testing individual units of code.

Q33. What is the blocking code?

Blocking code is code that cannot be executed until the current code is completely executed.

Q34. Are there any disadvantages of Node.js?

No technology comes without a few disadvantages. Node.js also has a few drawbacks. The main drawback is that Node.js responses can be greatly blocked if an intensive CPU computation is used.

In such cases, the multiple thread options are better but they are sluggish in performance. Moreover, if a relational database is used with Node.js, it behaves strangely, preventing favorable outcomes that the users can be sure of. Since Node.js does not support multiple threads, it is better suited for lightweight applications but not large-scale or heavy applications.

When using a relational database with Node.js, issues might arise due to the fundamental differences between the nature of Node.js and relational databases.

Some common challenges include:

1. **Blocking Operations**: Node.js is single-threaded and operates on a non-blocking, event-driven model. Traditional relational databases, however, typically use synchronous, blocking operations. This mismatch can lead to performance issues if queries are complex, causing them to “block” the main thread and preventing Node.js from handling other tasks effectively.

2. **Connection Management**: Node.js applications often need to maintain a large number of concurrent database connections, which relational databases may struggle with if not configured for such scale. This can result in connection pooling issues, slow queries, and timeouts.
3. **Concurrency and Transactions**: Handling transactions in Node.js with relational databases requires careful management, as Node.js does not handle multi-threading natively. This can lead to unexpected behaviors, such as race conditions or inconsistencies in database transactions if not managed properly.
4. **ORM Complexity**: Object-Relational Mapping (ORM) tools like Sequelize or TypeORM are often used to interface Node.js with relational databases, but they can add complexity. Misusing ORMs can lead to inefficient queries or data-fetching patterns that degrade performance.
5. **Schema Rigidity**: Relational databases have fixed schemas, while Node.js applications—especially in JavaScript—often benefit from flexible, schema-less data handling. This rigidity can make development slower and more error-prone if frequent schema changes are needed.

Q35. Give some examples of async functions.

Some examples of async functions are setTimeout(), setInterval(), find(), findOne(), save(), deleteOne fs.readFile(), fs.writeFile(), fs.appendFile(), readdir(), mkdir(), GET, PUT, POST, DELETE and process.nextTick()

Q36. Which library provides Node.js with a JavaScript engine?

The V8 library provides Node.js with the JavaScript engine.

Q37. How is JavaScript different from Node.js?

JavaScript is a programming language, whereas Node.js is an interpretation and environment for JavaScript. Node.js is used for performing non-blocking operations of any operating system. On the other hand, JavaScript is used for comprehensive application development.

Comprehensive application development refers to the process of creating an application that is fully functional, robust, and meets all intended requirements and objectives. It involves addressing every aspect of the application's lifecycle, from planning and design to deployment and maintenance. The goal is to ensure that the application is complete, well-rounded, and capable of serving its purpose effectively in real-world scenarios.

Q38. What are some features of Node.js?

It is fast, scalable, open-source, and asynchronous.

Q39. Can you develop network applications with Node.js?

Yes, Node.js developers can develop a variety of applications, including network applications.

A **network application** is a software application designed to operate across a network, enabling communication and interaction between users, devices, or systems. It relies on network protocols (e.g., TCP/IP, HTTP, WebSocket) to facilitate data exchange over the internet or local area networks (LANs).

Q40. What tasks can be done asynchronously with the help of an event loop?

The tasks include - intensive CPU computation, I/O operations, GUI programming, and database operations can be done asynchronously with the help of an event loop.

Q41. File system in Node js.

The **File System (fs) module** in Node.js allows developers to interact with the file system of the operating system. It provides methods to create, read, update, delete, and manipulate files and directories.

---

1. Importing the fs Module

To use the file system module, you must require it:

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

---

2. Key Categories of File System Operations

a. Reading Files

- **Asynchronous:**

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

- **Synchronous:**

```
const data = fs.readFileSync('example.txt', 'utf8');
console.log(data);
```

---

b. Writing Files

- **Asynchronous:**

```
fs.writeFile('example.txt', 'Hello, Node.js!', (err) => {
  if (err) {
    return console.error(err);
  }
  console.log('File written successfully');
});
```

- **Synchronous:**

```
fs.writeFileSync('example.txt', 'Hello, Node.js!');
console.log('File written successfully');
```

---

c. Appending to Files

- **Asynchronous:**

```
fs.appendFile('example.txt', '\nAppended text!', (err) => {
  if (err) {
    return console.error(err);
  }
  console.log('Text appended successfully');
});
```

- **Synchronous:**

```
fs.appendFileSync('example.txt', '\nAppended text!');
console.log('Text appended successfully');
```

---

d. Deleting Files

- **Asynchronous:**

```
fs.unlink('example.txt', (err) => {
  if (err) {
    return console.error(err);
  }
  console.log('File deleted successfully');
});
```

- **Synchronous:**

```
fs.unlinkSync('example.txt');
console.log('File deleted successfully');
```

---

e. Creating Directories

- **Asynchronous:**

```
fs.mkdir('newDir', { recursive: true }, (err) => {
  if (err) {
    return console.error(err);
  }
  console.log('Directory created successfully');
});
```

- **Synchronous:**

```
fs.mkdirSync('newDir', { recursive: true });
console.log('Directory created successfully');
```

---

f. Reading Directories

- **Asynchronous:**

```
fs.readdir('someDir', (err, files) => {
  if (err) {
    return console.error(err);
  }
  console.log('Files in directory:', files);
});
```

- **Synchronous:**

```
const files = fs.readdirSync('someDir');
console.log('Files in directory:', files);
```

---

g. Checking File/Directory Existence

- **Using fs.existsSync:**

```
if (fs.existsSync('example.txt')) {
  console.log('File exists');
} else {
  console.log('File does not exist');
}
```

- **Using fs.access:**

```
fs.access('example.txt', fs.constants.F_OK, (err) => {
  console.log(err ? 'File does not exist' : 'File exists');
});
```

---

3. Working with Streams

File System streams are used for reading and writing large files more efficiently.

Reading Streams:

```
const readStream = fs.createReadStream('example.txt', 'utf8');
readStream.on('data', (chunk) => {
  console.log(chunk);
});
```

Writing Streams:

```
const writeStream = fs.createWriteStream('output.txt');
writeStream.write('This is a stream write.\n');
writeStream.end();
```

Piping Streams (Reading and Writing Combined):

```
const readStream = fs.createReadStream('example.txt', 'utf8');
const writeStream = fs.createWriteStream('output.txt');
readStream.pipe(writeStream);
```

---

4. File Modes

Node.js supports file modes for managing permissions when working with files:

- **r**: Open file for reading.
- **w**: Open file for writing (overwrites if it exists).
- **a**: Open file for appending.

---

5. Error Handling

It's important to handle errors while working with the file system:

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

Common Use Cases

- Building file upload/download features.
- Creating configuration files.
- Managing logs.
- Reading or writing data for file-based storage.

The fs module is a powerful tool in Node.js for handling all types of file and directory operations, enabling developers to create efficient and scalable applications.

How to open file in Nodejs

To open a file in Node.js, you can use the fs.open method. This method opens a file descriptor, allowing you to interact with the file for reading, writing, or both.

---

fs.open Syntax

fs.open(path, flags, [mode], callback)

Parameters

1. **path**:
   - The path to the file to be opened (absolute or relative).
2. **flags**:
   - Specifies how the file should be opened.
   - Examples:
     - 'r': Open for reading (throws an error if the file doesn't exist).
     - 'r+': Open for reading and writing.
     - 'w': Open for writing (creates the file if it doesn't exist or truncates it).
     - 'a': Open for appending.
3. **mode** (optional):
   - File permission settings if the file is created (default is 0o666).
4. **callback**:
   - A function that gets executed when the file is opened. It has the signature:

(err, fd) => {...}
▪ err: An error object (if any).
▪ fd: The file descriptor, used for further operations on the file.

Example 1: Opening a File for Reading

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

fs.open('example.txt', 'r', (err, fd) => {
  if (err) {
    return console.error('Error opening file:', err.message);
  }
  console.log('File opened successfully:', fd);

  // Close the file descriptor
  fs.close(fd, (err) => {
    if (err) console.error('Error closing file:', err.message);
    else console.log('File closed successfully');
  });
});
```

Example 2: Opening a File for Writing

```
fs.open('example.txt', 'w', (err, fd) => {
  if (err) {
    return console.error('Error opening file:', err.message);
  }
  console.log('File opened for writing:', fd);

  // Writing to the file
  fs.write(fd, 'Hello, Node.js!', (err) => {
    if (err) console.error('Error writing to file:', err.message);
    else console.log('Write successful');

    // Close the file
    fs.close(fd, (err) => {
      if (err) console.error('Error closing file:', err.message);
      else console.log('File closed successfully');
    });
  });
});
```

Using fs.promises.open

The fs.promises.open method allows you to work with promises (ideal for async/await).

Example:

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

(async () => {
  try {
    const fileHandle = await fs.open('example.txt', 'r');
    console.log('File opened successfully');

    // Close the file
    await fileHandle.close();
    console.log('File closed successfully');
  } catch (err) {
    console.error('Error:', err.message);
  }
})();
```

Common File Open Flags

Flag	Description
'r'	Open for reading. File must exist.
'r+'	Open for reading and writing. File must exist.
'w'	Open for writing. Creates the file or truncates it.
'w+'	Open for reading and writing. Creates the file.
'a'	Open for appending. Creates the file if it doesn't exist.

Flag	Description
'a+'	Open for reading and appending. Creates the file.

Notes

- Always **close file descriptors** after you're done to prevent resource leaks. Use fs.close or the .close() method on a FileHandle.
- For simpler operations like reading or writing, consider using fs.readFile, fs.writeFile, or fs.appendFile directly, which handle file opening and closing for you.

Q42. What is Package.json?

Present in the root directory of a Node application/module, package.json defines the properties of a package including dependencies, metadata, and configuration options.

Q43. difference between fs.mkdir and fs.mkdirSync

The difference between fs.mkdir and fs.mkdirSync in Node.js lies in how they handle **asynchronous** and **synchronous** operations for creating directories.

---

fs.mkdir:

- **Type**: Asynchronous (non-blocking)
- **Behavior**:
  - Does not block the execution of the program.
  - Takes a callback function that is invoked once the directory is created or if there is an error.
- **Use Case**: Recommended for scenarios where blocking the execution of other operations should be avoided, especially in server environments.

Example:

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

fs.mkdir('exampleDir', { recursive: true }, (err) => {
  if (err) {
    return console.error('Error creating directory:', err);
  }
  console.log('Directory created successfully');
});

console.log('This runs immediately after calling fs.mkdir');
```

Output:

```
This runs immediately after calling fs.mkdir
Directory created successfully
```

---

fs.mkdirSync:

- **Type**: Synchronous (blocking)
- **Behavior**:

- Blocks the execution of subsequent code until the directory is created.
- Throws an error if the directory cannot be created (must use try-catch for error handling).

- **Use Case**: Suitable for scripts or tools where blocking operations are acceptable or desired.

Example:

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

try {
  fs.mkdirSync('exampleDir', { recursive: true });
  console.log('Directory created successfully');
} catch (err) {
  console.error('Error creating directory:', err);
}

console.log('This runs after fs.mkdirSync');
```

Output:

```
Directory created successfully
This runs after fs.mkdirSync
```

Key Differences

Feature	fs.mkdir (Asynchronous)	fs.mkdirSync (Synchronous)
Blocking Behavior	Non-blocking	Blocking
Callback	Requires a callback for results	Returns directly or throws error
Error Handling	Errors are passed to the callback	Errors must be caught explicitly
Performance Impact	Better for high-performance tasks	Can cause delays in execution

---

Choosing Between Them

- Use fs.mkdir for non-blocking operations in server environments or real-time applications.
- Use fs.mkdirSync in simple scripts or when the program logic requires the directory to be created before proceeding further.

Q44. What are the clauses used in promise object in Node.js?

In JavaScript, a Promise object can have three states:

Pending: The initial state of the promise before it is resolved or rejected.

Fulfilled: The state of a promise representing a successful operation. This is also sometimes called "resolved."

Rejected: The state of a promise representing a failed operation.

To create a Promise object, you must pass a function (often called an executor function) to the Promise constructor.

This function takes two arguments: resolve and reject. These are functions that you call to either fulfill or reject the promise.

Q45. Name some important applications in IT where Node.js can be used.

One of the main applications where Node.js can be used is building real-time web applications. Apart from this, distributed systems for sub-programming collections can also use Node.js. It can be used in general applications as well as complex network applications. One can also use Node.js to create, read, write, or close server files.

Q46. Is AJAX supported by all browsers?

Yes, all browsers support AJAX.

AJAX stands for **Asynchronous JavaScript and XML**. It is a technique used in web development to send and receive data from a server asynchronously without requiring the entire web page to reload. AJAX enhances user experience by enabling dynamic and interactive web applications.

Ajax is a client-side technology that allows for asynchronous communication between the client and the server. It is typically used to update parts of a web page without requiring a full page reload.

Key Characteristics of AJAX:

1. **Asynchronous Communication**: Allows parts of a web page to update independently without reloading the entire page.
2. **Server Interaction**: Facilitates seamless communication between the browser (client-side) and the server.
3. **Cross-Technology Approach**: Combines several web technologies (JavaScript, XML/JSON, HTML, and CSS) to provide a smooth user experience.

How AJAX Works:

The AJAX workflow typically involves the following steps:

1. **User Action**: A user action (like clicking a button) triggers an event on the web page.
2. **AJAX Request**: JavaScript sends an HTTP request to the server using the XMLHttpRequest object or modern fetch API.
3. **Server Response**: The server processes the request and sends back a response, usually in JSON or XML format.
4. **Update Page**: JavaScript processes the server's response and dynamically updates the web page without a full reload.

Modern Alternatives:

- AJAX is still widely used, but newer tools and frameworks, such as **Axios**, **GraphQL**, and real-time communication protocols like **WebSocket**, are becoming popular for more complex or modern web applications.

Q47. How to obtain the IP address of the user in Node.js?

We use req.connection.remote address to get the IP address.

Q48. How to install the Node body-parser module?

body-parser is a middleware module in Node.js that parses incoming request bodies in HTTP requests and makes the parsed data available under the req.body object. It is commonly used in applications that handle form submissions or JSON payloads.

Why Use body-parser?

1. HTTP requests have a body, but by default, Node.js does not parse the request body into a usable format.
2. body-parser helps convert the request body into a JavaScript object or other desired format so it can be processed easily.
3. It supports parsing various content types, including:
    - application/json
    - application/x-www-form-urlencoded
    - text/plain
    - multipart/form-data (usually handled by another module like multer).

To install the body-parser module for Node.js, follow these steps:

- Open a command prompt or terminal window.
- Navigate to the directory where your Node.js project is located.
- Run the command “npm install body-parser” to install the body-parser module and add it to your project's dependencies.

Q49. What is the difference between Angular and Node.js?

Angular	Node.js
It is a frontend development framework	It is a server-side environment
It is written in TypeScript	It is written in C, C++ languages
Used for building single-page, client-side web applications	Used for building fast and scalable server-side networking applications
Splits a web application into MVC components	Generates database queries

Q50. Which database is more popularly used with Node.js?

MongoDB is the most common database used with Node.js. It is a NoSQL, cross-platform, document-oriented database that provides high performance, high availability, and easy scalability.

Document-oriented refers to a type of database design that uses **documents** as the primary unit of data storage and organization. These documents are typically stored in formats like **JSON**, **BSON** (Binary JSON), or **XML**, making it easier to handle complex and hierarchical data structures.

This approach is a key feature of **NoSQL databases**, such as **MongoDB**, **CouchDB**, and **Firebase Firestore**.

Q51. What are some of the most commonly used libraries in Node.js?

There are two commonly used libraries in Node.js:

- ExpressJS - Express is a flexible Node.js web application framework that provides a wide set of features to develop web and mobile applications.
- Mongoose - Mongoose is also a Node.js web application framework that makes it easy to connect an application to a database.

Q52. What are the pros and cons of Node.js?

Node.js Pros	Node.js Cons
Fast processing and an event-based model	Not suitable for heavy computational tasks
Uses JavaScript, which is well-known amongst developers	Using callback is complex since you end up with several nested callbacks
Node Package Manager has over 50,000 packages that provide the functionality to an application	Dealing with relational databases is not a good option for Node.js
Best suited for streaming huge amounts of data and I/O intensive operations	Since Node.js is single-threaded, CPU intensive tasks are not its strong suit

Q53. What is the command used to import external libraries?

The “require” command is used for importing external libraries. For example - “var http=require (“HTTP”).” This will load the HTTP library and the single exported object through the HTTP variable.

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

Q54. Differentiate between process.nextTick() and setImmediate()?

The distinction between method and product. This is accomplished through the use of nextTick() and setImmediate(). nextTick() postpones the execution of action until the next pass around the event loop, or it simply calls the callback function once the event loop's current execution is complete, whereas setImmediate() executes a callback on the next cycle of the event loop and returns control to the event loop for any I/O operations.

The primary difference between process.nextTick() and setImmediate() in Node.js lies in **when** they execute in the event loop. Here's a detailed comparison:

Execution Timing

1. **process.nextTick()**
    - Executes **before the next phase of the event loop** begins.
    - It adds the callback to the **Next Tick Queue**, which is prioritized over other phases in the event loop.
    - Used for deferring a task to execute **immediately after the current operation**, without waiting for the next iteration of the event loop.
2. **setImmediate()**
    - Executes in the **check phase** of the event loop.
    - It schedules the callback to execute **after the I/O events and timers phase**, in the next iteration of the event loop.

Q55. What is an EventEmitter in Node.js?

- EventEmitter is a class that holds all the objects that can emit events
- Whenever an object from the EventEmitter class throws an event, all attached functions are called upon synchronously.

```
const EventEmitter = require('events');
class MyEmitter extends EventEmitter { }
const myEmitter = new MyEmitter();
myEmitter.on('event', () => {
console.log('an event occurred!');
});
myEmitter.emit('event');
```

Q56. What are the two types of API functions in Node.js?

The two types of API functions in Node.js are:

- Asynchronous, non-blocking functions
- Synchronous, blocking functions

Q57. What is the package.json file?

The package.json file is the heart of a Node.js system. This file holds the metadata for a particular project. The package.json file is found in the root directory of any Node application or module

This is what a package.json file looks like immediately after creating a Node.js project using the command: npm init

You can edit the parameters when you create a Node.js project.

```
{
  "name": "node-npm",
  "version": "1.0.0",
  "description": "A demo application",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Taha",
  "license": "ISC"
}
```

Q58. How would you use a URL module in Node.js?

The URL module in Node.js provides various utilities for URL resolution and parsing. It is a built-in module that helps split up the web address into a readable format.

```
var url = require('url');
var adrs = 'http://localhost:8080/default.htm?year=2020&
month=march';
var que = url.parse(adrs, true);
console.log(que.host); //returns 'localhost:8080'
console.log(que.pathname); //returns '/default.htm'
console.log(que.search); //returns '?year=2020 and month
=march'
var quedata = que.query; //returns an object: { year: 2020,
 month: 'march' }
console.log(quedata.month); //returns 'march'
```

Q59. What is the Express.js package?

Express is a flexible Node.js web application framework that provides a wide set of features to develop both web and mobile applications

Q60. How do you create a simple Express.js application?

- The request object represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, and so on
- The response object represents the HTTP response that an Express app sends when it receives an HTTP request

```
import express from 'express'
const app = express(); // Create an Express application
const port = process.env.PORT || "8000";
const DATABASE_URL = process.env.DATABASE_URL || "mongodb://127.0.0.1:27017"
// Start the server
server.listen(port, () => {
    console.log(`server listening at http://localhost:${port}`)
})
```

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

app.get('/', function (req, res) {
  res.send('Hello World');
})

var server = app.listen(8081, function () {
  var host = server.address().address
  var port = server.address().port

  console.log("Example app listening at http://%s:%s", host, port)
})
```

The code provided in the image is an example of a basic Express.js server setup. Let me break it down and explain the concepts, particularly focusing on the **request** (req) and **response** (res) objects.

Code Explanation

1. **Importing Express and Creating an Application:**

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

   - The express module is imported, and an Express application (app) is created. This app is the main object for handling HTTP requests and responses.

2. **Defining a Route:**

```
app.get('/', function (req, res) {
  res.send('Hello World');
});
```

   - **app.get('/')**: Sets up a route to handle HTTP GET requests made to the root URL (/).
   - **req (Request Object)**: Represents the incoming HTTP request and contains details such as query strings, parameters, headers, and body.
   - **res (Response Object)**: Represents the HTTP response that the server will send back to the client.
     - In this case, res.send('Hello World') sends the string "Hello World" as the response body.

3. **Starting the Server:**

```
var server = app.listen(8081, function () {
  var host = server.address().address;
  var port = server.address().port;
  console.log('Example app listening at http://%s:%s', host, port);
});
```

   - **app.listen(8081)**: Starts the server on port 8081.
   - **server.address()**: Returns an object containing information about the server, including its address and port.
   - The console logs the URL where the server is accessible.

Request (req) Object

The req object contains all the information about the client's HTTP request. Here are some commonly used properties:

- **req.query**: Contains query parameters in the URL.
    - Example: For http://localhost:8081/?name=John, req.query.name would be John.
- **req.params**: Contains route parameters (e.g., /user/:id).
    - Example: For /user/123, req.params.id would be 123.
- **req.body**: Contains the request payload, typically for POST/PUT requests.
    - Requires middleware like body-parser to parse JSON or form data.
- **req.headers**: Contains HTTP headers sent by the client.
    - Example: req.headers['content-type'] gives the Content-Type header.

Response (res) Object

The res object is used to send a response back to the client. Common methods include:

- **res.send()**: Sends a response (can be a string, object, or array).
- **res.json()**: Sends a JSON response.
- **res.status()**: Sets the HTTP status code.
    - Example: res.status(404).send('Not Found').
- **res.redirect()**: Redirects the client to another URL.

Example Use Case

- When a user visits http://localhost:8081/, the Express server will:
    - Receive the request and create a req object containing details about the request.
    - Process the route logic and use the res object to send back "Hello World" as a response.

This structure allows developers to handle incoming requests and send tailored responses efficiently in web applications.,

Q61. What are streams in Node.js?

Streams are objects that enable you to read data or write data continuously.

There are four types of streams:

Readable – Used for reading operations

Writable – Used for write operations

Duplex – Can be used for both reading and write operations

Transform – A type of duplex stream where the output is computed based on input

Q62. How do you install, update, and delete a dependency?

Q63. How do you create a simple server in Node.js that returns Hello World?

- Import the HTTP module
- Use createServer function with a callback function using request and response as parameters.
- Type “hello world."
- Set the server to listen to port 8080 and assign an IP address

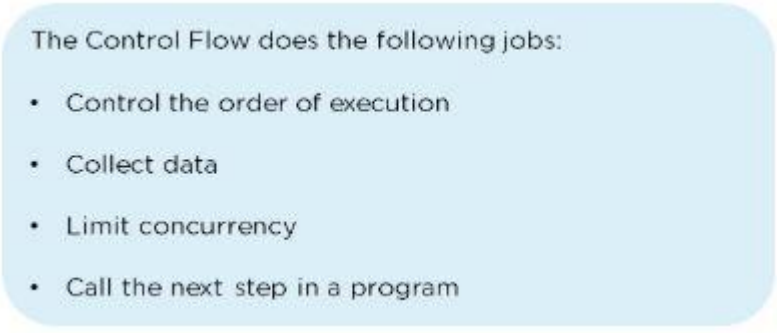
Q64. Explain asynchronous and non-blocking APIs in Node.js.

- All Node.js library APIs are asynchronous, which means they are also non-blocking
- A Node.js-based server never waits for an API to return data. Instead, it moves to the next API after calling it, and a notification mechanism from a Node.js event responds to the server for the previous API call.

Q65. What is the control flow function?

The control flow function is a piece of code that runs in between several asynchronous function calls.

Q66. How does control flow manage the function calls?



Q67. What is the difference between fork() and spawn() methods in Node.js?

fork()	spawn()

`child_process.fork(modulePath[, args][, options])`	`child_process.spawn(command[, args][, options])`
fork() is a particular case of spawn() that generates a new instance of a V8 engine.	Spawn() launches a new process with the available set of commands.
Multiple workers run on a single node code base for multiple tasks.	This method doesn't generate a new V8 instance, and only a single copy of the node module is active on the processor.

Q68. What is the buffer class in Node.js?

Buffer class stores raw data similar to an array of integers but corresponds to a raw memory allocation outside the V8 heap. Buffer class is used because pure JavaScript is not compatible with binary data.

Q69. What is piping in Node.js?

Piping is a mechanism used to connect the output of one stream to another stream. It is normally used to retrieve data from one stream and pass output to another stream.

Q70. What are some of the flags used in the read/write operations in files?

- **r** – Open file for reading. An exception occurs if the file does not exist.
- **r+** – Open file for reading and writing. An exception occurs if the file does not exist.
- **w** – Open file for writing. The file is created (if it does not exist) or truncated (if it exists).
- **w+** – Open file for reading and writing. The file is created (if it does not exist) or truncated (if it exists).
- **a** – Open file for appending. The file is created if it does not exist.
- **a+** – Open file for reading and appending. The file is created if it does not exist.

Q71. How do you open a file in Node.js?

This is the syntax for opening a file in Node.js

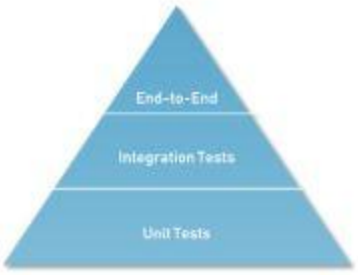
```
fs.open(path, flags[, mode], callback)
```

Q72. What is a reactor pattern in Node.js?

A reactor pattern is a concept of non-blocking I/O operations. This pattern provides a handler that is associated with each I/O operation. As soon as an I/O request is generated, it is then submitted to a demultiplexer

Q73. What is a test pyramid in Node.js?

- A test pyramid is a figure which explains the proportion of unit tests, integrations tests, and end-to-end tests that are required for the proper development of a project
- The components of a test pyramid are given below:
  - **Unit Tests:** They test the individual units of code in isolation
  - **Integrations Tests:** They test the integration among dissimilar units
  - **End-to-End (E2E) Tests:** They test the whole system, from the User Interface to the data store, and back.



Q74. For Node.js, why does Google use the V8 engine?

The V8 engine, developed by Google, is open-source and written in C++. Google Chrome makes use of this engine. V8, unlike the other engines, is also utilized for the popular Node.js runtime. V8 was initially intended to improve the speed of JavaScript execution within web browsers. Instead of employing an interpreter, V8 converts JavaScript code into more efficient machine code to increase performance. It turns JavaScript code into machine code during execution by utilizing a JIT (Just-In-Time) compiler, as do many current JavaScript engines such as SpiderMonkey or Rhino (Mozilla).

Q75. Describe Node.js exit codes.

Exit codes are a set of specific codes which are used for finishing a specific process. Given below are some of the exit codes used in Node.js:

- Uncaught fatal exception
- Unused
- Fatal Error
- Internal Exception handler Run-time failure
- Internal JavaScript Evaluation Failure

Q76. Explain the concept of middleware in Node.js.

Middleware is a function that receives the request and response objects. Most tasks that the middleware functions perform are:

- Execute any code
- Update or modify the request and the response objects
- Finish the request-response cycle
- Invoke the next middleware in the stack

Q77. What are the different types of HTTP requests?

HTTP defines a set of request methods used to perform desired actions. The request methods include:

GET: Used to retrieve the data

POST: Generally used to make a change in state or reactions on the server

HEAD: Similar to the GET method, but asks for the response without the response body

DELETE: Used to delete the predetermined resource

Q78. How would you connect a MongoDB database to Node.js?

To create a database in MongoDB:

- Start by creating a MongoClient object
- Specify a connection URL with the correct IP address and the name of the database you want to create

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/mydb";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  console.log("Database created!");
  db.close();
});
```

```
// dotenv should be import at top
import dotenv from 'dotenv'
dotenv.config()
import express from 'express';
import connectDB from './db/connectdb.js';

const app = express(); // Create an Express application
const port = process.env.PORT || "8000";
const DATABASE_URL = process.env.DATABASE_URL || "mongodb://127.0.0.1:27017"

// connecting mongodb
connectDB(DATABASE_URL)

const server = http.createServer(app);
// Start the server
server.listen(port, () => {
    console.log(`server listening at http://localhost:${port}`)
})
```

Q79. What is the purpose of NODE_ENV?

Q80. List the various Node.js timing features.

Timers module is provided by Node.js which contains various functions for executing the code after a specified period of time. Various functions that are provided by this module:

setTimeout/clearTimeout - Used to schedule code execution after a designated amount of milliseconds

setInterval/clearInterval - Used to execute a block of code multiple times

setImmediate/clearImmediate - Used to execute code at the end of the current event loop cycle

Q81. What is WASI, and why is it being introduced?

The WASI class implements the WASI system called API and extra convenience methods for interacting with WASI-based applications. Every WASI instance represents a unique sandbox environment. Each WASI instance must specify its command-line parameters, environment variables, and sandbox directory structure for security reasons.

Q82. What is a first-class function in Javascript?

First-class functions are a powerful feature of JavaScript that allows you to write more flexible and reusable code. In Node.js, first-class functions are used extensively in asynchronous programming to write non-blocking code.

Q83. How do you manage packages in your Node.Js project?

Managing packages in your Node.js project is done using the Node Package Manager (NPM), which allows you to install and manage third-party packages and create and publish your packages.

Q84. How is Node.js better than other frameworks?

Node.js is a server-side JavaScript runtime environment built on top of the V8 JavaScript engine, the same engine that powers Google Chrome. It makes Node.js very fast and efficient, as well as highly scalable.

Q85. What is a fork in node JS?

The Fork method in Node.js creates a new child process that runs a separate Node.js instance and can be useful for running CPU-intensive tasks or creating a cluster of Node.js servers.

Q86. List down the two arguments that async. First, does the queue take as input?

The async.queue function in Node.js takes two arguments as input: a worker function and an optional concurrency limit. It is used to create a task queue executed in parallel.

Q87. What is the purpose of the module.exports?

The module. exports object in Node.js is used to export functions, objects, or values from a module and is returned as the value of the require() function when another module requires a module.

Q88. What tools can be used to assure consistent code style?

In summary, several tools can be used in Node.js to ensure consistent code style and improve code quality, including ESLint, Prettier, and Jest.

Q89. What is the difference between JavaScript and Node.js?

Node.js is a runtime environment for executing JavaScript code outside of a web browser, while JavaScript is a programming language that can be executed in both web browsers and Node.js environments.

Q90. What is the difference between asynchronous and synchronous functions?

Synchronous functions block the execution of other code until they are complete, while asynchronous functions allow other code to continue executing while they are running, making them essential for writing scalable Node.js applications.

Q91. What are the asynchronous tasks that should occur in an event loop?

Asynchronous tasks that should occur in an event loop in Node.js include I/O operations, timers, and callback functions. By performing these tasks asynchronously, Node.js can handle a large number of concurrent requests without blocking the event loop.

Q92. What is the order of execution in control flow statements?

In Node.js, control flow statements are executed in a specific order. The order of execution is determined by the event loop. The event loop is a mechanism in Node.js that allows for the execution of non-blocking I/O operations.

Q93. What are the input arguments for an asynchronous queue?

An asynchronous queue in Node.js is a data structure that allows for the execution of functions in a specific order. Functions are added to the queue and are executed in the order that they were added. An asynchronous queue is useful when you want to execute a series of functions in a specific order.

Q94. Are there any disadvantages to using Node.js?

Node.Js is not suitable for CPU-intensive tasks. This is because Node.js is single-threaded, meaning it can only execute one task at a time. Node.js is not suitable for applications that require a lot of memory. This is because Node.js uses a lot of memory for each connection. If you have a large number of connections, it can quickly consume a lot of memory.

Q95. What is the primary reason for using the event-based model in Node.js?

The main reason to use the event-based model in Node.js is performance. The event-based model allows for non-blocking I/O operations, which means that Node.js can handle a large number of connections without using a lot of resources.

Q96. What is the difference between Node.js and Ajax?

Ajax and Node.js are two different technologies that are used for different purposes. Ajax is a client-side technology that allows for asynchronous communication between the client and the server. It is typically used to update parts of a web page without requiring a full page reload.

Node.js, on the other hand, is a server-side technology that is used for building fast, scalable, and efficient server-side applications. It is typically used for real-time applications, such as chat applications, online games, and streaming services.

Q97. What is the advantage of using Node.js?

Node.js is fast and scalable. Node.js is easy to learn and use. Node.js is well-suited for real-time applications, such as chat applications, online games, and streaming services. This is because Node.js can handle a large number of connections and can perform non-blocking I/O operations, which makes it ideal for real-time communication.

Q98. Does Node run on Windows?

Yes, Node.js runs on Windows. Node.js is a cross-platform runtime environment, which means that it can run on a variety of operating systems, including Windows, macOS, and Linux.

Q99. Can you access DOM in Node?

No, you cannot access the DOM in Node.js. The DOM is a browser-specific API that allows for the manipulation of HTML and XML documents. Since Node.js does not run in a browser, it does not have access to the DOM.

Q100. Why is Node.JS quickly gaining attention from JAVA programmers?

Node.js is quickly gaining attention from Java programmers because it is fast, scalable, and efficient. Java is a popular server-side technology, but it can be slow and resource-intensive. Node.js, on the other hand, is built on the V8 JavaScript engine, which is known for its speed and performance.

Q101. What are the Challenges with Node.js?

Node.js is single-threaded, which means that it can only execute one task at a time. Node.js is relatively new compared to other server-side technologies, such as Java and PHP. This means that there needs to be more support and more resources available for Node.js. Node.js is only suitable for applications that require a little memory.

Q102. How does Node.js overcome the problem of blocking I/O operations?

Node.js uses an event-driven, non-blocking I/O model that allows it to handle I/O operations more efficiently. By using callbacks, Node.js can continue processing other tasks while waiting for I/O operations to complete. This means that Node.js can handle multiple requests simultaneously without causing any delays. Additionally, Node.js uses a single-threaded event loop architecture, which allows it to handle a high volume of requests without any issues.

Q103. How can we use async await in node.js?

To use async/await in Node.js, you'll need to use functions that return promises. You can then use the async keyword to mark a function as asynchronous and the await keyword to wait for a promise to resolve before continuing with the rest of the code.

Q104. Why should you separate the Express app and server?

Firstly, separating your app and server can make it easier to test your code. By separating the two, you can test your app logic independently of the server, which can make it easier to identify and fix bugs.

Secondly, separating your app and server can make it easier to scale your application. By separating the two, you can run multiple instances of your app on different servers, which can help to distribute the load and improve performance.

Finally, separating your app and server can make it easier to switch to a different server if necessary. By keeping your app logic separate from your server logic, you can switch to a different server without having to make any major changes to your code.

Q105. Explain the concept of stub in Node.js.

In Node.js, a stub is a function that serves as a placeholder for a more complex function. Stubs are typically used in unit testing to replace a real function with a simplified version that returns a predetermined value. By using a stub, you can ensure that your unit tests are predictable and consistent.

Q106. What is the framework that is used majorly in Node.js today?

There are many frameworks available for Node.js, but the two most popular ones are Express and Koa.

Q107. What are the security implementations that are present in Node.js?

One of the most important security features in Node.js is the ability to run code in a restricted environment. This is achieved through the use of a sandboxed environment, which can help to prevent malicious code from accessing sensitive data or causing any damage to the system.

Another important security feature in Node.js is the ability to use TLS/SSL to encrypt data in transit. This can help to prevent eavesdropping and ensure that sensitive data is protected.

Q108. What is Libuv?

Libuv is a critical component of Node.js, and it's what makes it possible to handle I/O operations in a non-blocking and efficient manner.

Q109. What are global objects in Node.js?

Global objects in Node.js are objects that are available in all modules without the need for an explicit require statement. Some of the most commonly used global objects in Node.js include process, console, and buffer.

Q110. what is middleware

Middleware in Node.js (specifically in the **Express framework**) refers to **functions that execute during the request-response cycle**. Middleware functions have access to the req (request), res (response), and the next function in the application's workflow. They are used to process requests, modify responses, or execute additional logic before sending a response back to the client.

Key Features of Middleware

1. **Access to Request and Response Objects**: Middleware can modify req and res objects.
2. **Chain of Execution**: Middleware functions are executed sequentially.
3. **next() Function**: Middleware passes control to the next middleware function using next().

Types of Middleware

1. Application-Level Middleware

- Defined at the app level.
- Executes for all or specific routes.
- Example:

```
const express = require('express');
const app = express();
// Middleware function
app.use((req, res, next) => {
  console.log(`Request Method: ${req.method}, URL: ${req.url}`);
  next(); // Pass control to the next middleware
});

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

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

2. Router-Level Middleware

- Associated with specific routers using router.use() or route handlers.
- Example:

```
const express = require('express');
const router = express.Router();

// Router middleware
router.use((req, res, next) => {
  console.log('Router-level middleware triggered');
  next();
});
```

```
router.get('/users', (req, res) => {
  res.send('User list');
});

const app = express();
app.use('/api', router);
app.listen(3000, () => console.log('Server running on port 3000'));
```

3. Built-in Middleware

- Middleware provided by Express.
- Examples:
  - **express.json()**: Parses incoming JSON payloads.
    - **express.json()**

What It Does

- express.json() parses incoming JSON payloads from the request body.
- It is typically used when the client sends data in JSON format (e.g., in a POST request).
- After parsing, the JSON data is available as a JavaScript object in req.body.

Use Case

- API requests where data is sent in JSON format (commonly in RESTful APIs).

How It Works

1. The client sends a request with the Content-Type header set to application/json.
2. express.json() reads the payload, parses it into a JavaScript object, and attaches it to req.body.

Important Notes

- express.json() throws an error if the incoming JSON payload is malformed.
- Example of a malformed payload:

```
{ "name": "John Doe" // missing closing brace
app.use(express.json()); // Built-in middleware for parsing JSON
```

  - **express.urlencoded()**: Parses URL-encoded payloads.
    - **express.urlencoded()**

What It Does

- express.urlencoded() parses incoming requests with URL-encoded payloads.
- It is typically used when data is sent through **HTML forms** using the application/x-www-form-urlencoded format (default for form submissions).

Use Case

- Handling form submissions where data is sent in key-value pairs.

How It Works

1. The client sends a request with the Content-Type header set to application/x-www-form-urlencoded.
2. express.urlencoded() parses the key-value pairs and attaches them as a JavaScript object to req.body.

Key Differences Between express.json() and express.urlencoded()

Feature	express.json()	express.urlencoded()
Purpose	Parses JSON payloads.	Parses URL-encoded payloads (form data).
Content-Type Header	application/json	application/x-www-form-urlencoded
Use Case	For APIs that receive JSON data.	For handling HTML form submissions.
Payload Format	JSON objects (e.g., { "key": "value" })	Key-value pairs (e.g., key=value&key2=value2).
Extended Option	Not applicable.	Supports nested objects when extended: true.

4. Third-Party Middleware

- Middleware provided by external libraries.
- Examples:
    - **morgan**: Logs HTTP requests.
    - **cors**: Enables Cross-Origin Resource Sharing.

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

app.use(morgan('dev')); // Logs HTTP requests
```

What is CORS?

CORS (Cross-Origin Resource Sharing) is a **security feature** implemented in web browsers that governs how resources on a web server are shared with web pages from **different origins**. It defines a mechanism for allowing or restricting **cross-origin requests**.

Why is CORS needed?

Browsers enforce a security policy called the **Same-Origin Policy** (SOP), which restricts web pages from making requests to a different origin than the one that served the web page. An **origin** is defined by the combination of:

- **Protocol** (e.g., http, https)
- **Domain** (e.g., example.com)
- **Port** (e.g., :3000, :8080)

For example:

- A page served from http://example.com:3000 **cannot** access resources from http://another-domain.com without proper permissions.

CORS provides a way for the server to **relax this restriction** and specify which origins are allowed to access its resources.

5. Error-Handling Middleware

- Catches and processes errors.
- Must have 4 parameters: (err, req, res, next).
- Example:

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('Something went wrong!');
});
```

Common Use Cases for Middleware

1. **Logging**: Record details of incoming requests.

```
app.use((req, res, next) => {
  console.log(`Request received at ${new Date()}`);
  next();
});
```

2. **Authentication**: Verify user credentials before proceeding to the route.

```
app.use((req, res, next) => {
  if (!req.headers.authorization) {
    return res.status(403).send('Unauthorized');
  }
  next();
});
```

3. **Parsing Data**: Parse JSON, form data, or cookies from incoming requests.
4. **Static Files**: Serve static files like images, CSS, and JS.

```
app.use(express.static('public'));
```

5. **Error Handling**: Provide a centralized mechanism for handling errors.

Middleware Flow

1. A request hits the server.
2. Middleware functions are executed in the order they are defined.
3. If next() is called, the next middleware is executed.
4. If no middleware sends a response, the request eventually reaches the route handler.

Middleware Example with Complete Flow

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

// Middleware 1
app.use((req, res, next) => {
  console.log('Middleware 1: Request received');
  next(); // Pass control to the next middleware
```

```
});

// Middleware 2
app.use((req, res, next) => {
  console.log('Middleware 2: Processing request');
  next(); // Pass control to the next route or middleware
});

// Route Handler
app.get('/', (req, res) => {
  console.log('Route Handler: Sending response');
  res.send('Hello from Express!');
});

// Start the server
app.listen(3000, () => console.log('Server is running on port 3000'));
```

Output of the Example

1. Client sends a request to /.
2. Logs:

```
Middleware 1: Request received
Middleware 2: Processing request
Route Handler: Sending response
```

3. Response: **Hello from Express!**

This demonstrates how middleware acts as a chain of functions that process requests before they reach the final route handler.

Q111. Why is ExpressJS used?

Express is a great choice for building web applications in Node.js, and its popularity and active community make it a safe and reliable choice for developers of all levels.

Q112. What is the use of the connect module in Node.js?

The Connect module can be used to handle different types of middleware, such as error-handling middleware, cookie-parsing middleware, and session middleware. Error-handling middleware is used to handle errors that occur during the request/response cycle. Cookie parsing middleware is used to parse cookies from the request header. Session middleware is used to manage user sessions.

Q113. What's the difference between 'front-end' and 'back-end' development?

Front-end developers focus on the client side of the application, while back-end developers focus on the server side of the application. Both roles are important for building a successful web application and require different skill sets and expertise.

Q114. What are LTS releases of Node.js?

LTS stands for Long-term support. LTS releases of Node.js are versions that are supported for an extended period, usually for 30 months from the time of release. These releases are typically more stable and reliable than non-LTS releases and are recommended for production use.

Q115. What do you understand about ESLint?

ESLint is a popular open-source tool that is used to analyze and flag errors and potential problems in JavaScript code.

Q116. Define the concept of the test pyramid. Please explain the process of implementing them in terms of HTTP APIs.

The test pyramid is a concept that is often used in software testing to illustrate the ideal distribution of different types of tests. The pyramid consists of three layers: unit tests, integration tests, and end-to-end tests. The idea is that the majority of tests should be at the unit level, with fewer tests at the integration and end-to-end levels.

To implement the test pyramid in terms of HTTP APIs, you can start by writing unit tests for each endpoint in the API. These tests should focus on testing the functionality of the endpoint in isolation without making any external requests or dependencies. Once the unit tests are passed, you can write integration tests that test the interaction between different endpoints and components in the API. Finally, you can write end-to-end tests that test the entire API, from the user interface to the database.

Q117. How does Node.js handle the child threads?

Node.js handles child threads by creating separate instances of the Node.js runtime environment that can be used to execute code in parallel with the main process.

Q118. What is an Event Emitter in Node.js?

An Event Emitter is a Node.js module that facilitates communication between objects in a Node.js application. It is an instance of the EventEmitter class, which provides a set of methods to listen for and emit events. In Node.js, events are a core part of the platform, and they are used to handle asynchronous operations.

Q119. How to Enhance Node.js Performance through Clustering?

Clustering can be used to improve the performance of HTTP servers, database connections, and other I/O operations. However, it is important to note that clustering does not guarantee a linear increase in performance.

Q120. What is a thread pool, and which library handles it in Node.js?

A thread pool is a collection of threads that are used to execute tasks in parallel. In Node.js, the thread pool is handled by the libuv library, which is a multi-platform support library that provides asynchronous I/O operations.

Q121. How are worker threads different from clusters?

Worker threads and clusters are two different approaches to leveraging (using something to its full advantage or maximizing its potential to achieve a desired result) the power of multiple CPUs in Node.js. While clusters create multiple instances of a Node.js process, each running on a separate CPU core, worker threads provide a way to create multiple threads within a single process.

Q122. How to measure the duration of async operations?

The console.time and console.timeEnd methods allow you to measure the duration of a block of code. The console.time method is used to start the timer and the console.timeEnd method is used to stop the timer and log the duration to the console.

The performance.now method provides a more precise way to measure the duration of async operations. It returns the current timestamp in milliseconds, which can be used to calculate the duration of a task.

Q123. How to measure the performance of async operations?

There are several tools and techniques you can use to measure performance, including using the built-in --prof flag, using the perf tool, and using third-party libraries like benchmark.js.

Q124. What are the types of streams available in Node.js?

There are four types of streams available in Node.js, including readable streams, writable streams, duplex streams, and transform streams.

Q125. What is meant by tracing in Node.js?

Tracing is a technique used in Node.js to profile the performance of an application. It involves recording the function calls and events that occur during the execution of the application and analyzing the data to identify performance bottlenecks.

Q126. Where is package.json used in Node.js?

The package.json file is located in the root directory of an application and it is used by the npm package manager to install and manage the dependencies of an application.

Q127. What is the difference between readFile and create Read Stream in Node.js?

Create Read Stream is a better option for reading large files, while the read file is a better option for small files. It is important to choose the right method based on the size of the file and the requirements of the application.

Q128. What is the use of the crypto module in Node.js?

The crypto module is widely used in Node.js applications to generate secure random numbers, create digital signatures, and verify signatures. It also provides support for various encryption algorithms such as AES, DES, and RSA.

Q129. What is a passport in Node.js?

Passport is a popular authentication middleware for Node.js. It provides a simple and modular way to implement authentication in Node.js applications. Passport supports many authentication mechanisms, including username/password, social logins like Facebook and Google, and JSON Web Tokens (JWTs).

A **JSON Web Token (JWT)** is a compact, URL-safe token format used to securely transmit information between parties as a **JSON object**. It is commonly used for **authentication** and **authorization** in web applications.

A **JSON web token(JWT)** is *JSON Object* which is used to securely transfer information over the web(between two parties). It can be used for an authentication system and can also be used for information exchange.

Q130. How to get information about a file in Node.js?

In Node.js, the fs module provides methods for working with the file system. To get information about a file, you can use the fs. stat() method. The fs. stat() method returns an object that contains information about the file, such as the file size, creation date, and modified date.

Q131. How does the DNS lookup function work in Node.js?

In Node.js, the DNS module provides methods for performing DNS lookups. DNS stands for Domain Name System, and it is responsible for translating domain names into IP addresses. The DNS. lookup() method is used to perform a DNS lookup and resolve a domain name into an IP address.

Q132. What is the difference between setImmediate() and setTimeout()?

The setTimeout() method schedules code execution after a specified delay, measured in milliseconds. On the other hand, the setImmediate() method schedules code execution to occur immediately after the current event loop iteration completes. This means that setImmediate() has a higher priority than setTimeout().

Q133. Explain the concept of Punycode in Node.js.

Punycode is a character encoding scheme used in the domain name system (DNS) to represent Unicode characters with ASCII characters. It is used to encode domain names that contain non-ASCII characters, such as Chinese or Arabic characters.

Q134. Does Node.js provide any Debugger?

Yes, Node.js provides a built-in debugger that can be used to debug Node.js applications.

Q135. Is cryptography supported in Node.js?

Yes, Node.js provides built-in support for cryptography through the crypto module.

Q136. List async methods

Here is a list of commonly used asynchronous methods in Node.js, including built-in async functions and popular third-party ones that support async/await. These methods are commonly used for operations like file handling, network requests, database queries, and timing.

1. File System (fs.promises)

These methods from Node.js's fs.promises module allow asynchronous file operations.

- fs.promises.readFile(path, options): Reads the contents of a file.
- fs.promises.writeFile(file, data, options): Writes data to a file, replacing it if it already exists.
- fs.promises.appendFile(file, data, options): Appends data to a file.
- fs.promises.unlink(path): Deletes a file.
- fs.promises.readdir(path): Reads the contents of a directory.
- fs.promises.mkdir(path, options): Creates a new directory.
- fs.promises.rmdir(path): Removes a directory.

2. Networking (HTTP Requests)

HTTP request libraries like node-fetch, axios, and native fetch in supported environments are used to make async HTTP requests.

- **fetch(url, options)**: The native fetch API in some Node.js versions or environments.
- **axios.get(url, config)**: Asynchronous GET request.
- **axios.post(url, data, config)**: Asynchronous POST request.
- **axios.put(url, data, config)**: Asynchronous PUT request.
- **axios.delete(url, config)**: Asynchronous DELETE request.

3. Timers

Node.js has asynchronous timing methods for scheduling tasks.

- setTimeout(callback, ms): Executes a function after a specified delay (returns a Promise if wrapped).
- setInterval(callback, ms): Repeatedly executes a function at specified intervals (usually wrapped for async/await usage).
- setImmediate(callback): Executes a function asynchronously in the next event loop iteration.

4. Database (e.g., Mongoose, Sequelize)

Databases libraries like Mongoose for MongoDB or Sequelize for SQL databases provide async methods for querying and data manipulation.

Mongoose is a **Node.js framework/library** that provides a **MongoDB Object Data Modeling (ODM)** solution. It is specifically designed to work with MongoDB, making it easier to define schemas, interact with databases, and handle complex data operations in Node.js applications.

- **Mongoose (MongoDB)**:
    - Model.find(query): Finds documents in a collection.
    - Model.findOne(query): Finds a single document.
    - Model.save(): Saves a document.
    - Model.deleteOne(query): Deletes a document.

    - Model.updateOne(query, update): Updates a document.

- **Sequelize (SQL Databases)**:
    - Model.findAll(query): Finds all records.
    - Model.findOne(query): Finds a single record.
    - Model.create(data): Creates a new record.
    - Model.update(data, query): Updates records.
    - Model.destroy(query): Deletes records.

5. Process and Streams

Node.js supports asynchronous methods for handling data streams and process management.

- process.nextTick(callback): Executes code asynchronously in the next tick of the event loop.
- stream.readable.read(): Reads data from a readable stream asynchronously.
- stream.writable.write(chunk): Writes data to a writable stream asynchronously.

6. Crypto (for asynchronous encryption)

Node.js crypto module provides methods for performing cryptographic operations asynchronously.

- crypto.pbkdf2(password, salt, iterations, keylen, digest, callback): Derives a key from a password asynchronously.
- crypto.randomBytes(size, callback): Generates random bytes asynchronously.

7. Events and Promises

Node.js also has built-in utility functions for handling Promises.

- Promise.all(iterable): Runs multiple Promises concurrently and resolves when all Promises are complete.
- Promise.race(iterable): Resolves as soon as any of the Promises in the iterable resolves.
- Promise.allSettled(iterable): Resolves when all Promises have settled, regardless of success or failure.
- Promise.any(iterable): Resolves as soon as any Promise fulfills, ignoring rejected Promises.

8. Utilities

Utility modules also provide some asynchronous methods.

- util.promisify(function): Converts a callback-based function to return a Promise, making it compatible with async/await.

These async methods and libraries allow you to work with a wide range of asynchronous tasks in Node.js, from filesystem interactions to HTTP requests, database queries, and cryptographic operations.

Q137. What are security implementations within Node.js?

The different types of security implementations within Node.js include error handling, authentications and authorization, data sanitization, encryption, and logging and monitoring.

Q138. Why is assert used in Node.js?

An assert module is an important tool for writing effective tests in Node.js.

Q139. Routing in Nodejs.

Routing in Node.js refers to how an application handles client requests to specific URLs (or routes) with different HTTP methods (GET, POST, etc.). In Node.js, routing can be implemented using the **Express.js framework**, which simplifies the process.

Routing in Node.js with Express

1. **Install Express**:

   npm install express

2. **Basic Routing Example**:

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

// Define a route for GET requests to the root URL
app.get('/', (req, res) => {
  res.send('Welcome to the homepage!');
});

// Define a route for GET requests to /about
app.get('/about', (req, res) => {
  res.send('This is the about page.');
});

// Define a route for POST requests to /contact
app.post('/contact', (req, res) => {
  res.send('Contact form submitted!');
});

// Start the server
const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server is running on http://localhost:${PORT}`);
});
```

Route Parameters

Route parameters allow you to capture values from the URL and use them in your code.

```
app.get('/user/:id', (req, res) => {
  const userId = req.params.id;
  res.send(`User ID is ${userId}`);
});
```

- Visiting /user/123 will display User ID is 123.

Query Parameters

Query parameters are values passed in the URL after a ? symbol.

```
app.get('/search', (req, res) => {
```

```
  const query = req.query.q;
  res.send(`You searched for ${query}`);
});
```

- Visiting /search?q=nodejs will display You searched for nodejs.

Middleware in Routing

Middleware functions can be used in routes to perform tasks like authentication, logging, or data parsing.

```
// Middleware function
const logRequest = (req, res, next) => {
  console.log(`${req.method} request for ${req.url}`);
  next();
};

// Apply middleware to a route
app.get('/dashboard', logRequest, (req, res) => {
  res.send('Welcome to the dashboard!');
});
```

Chaining Route Handlers

You can chain multiple handlers for a single route.

```
app.get('/profile',
  (req, res, next) => {
    console.log('Handler 1');
    next(); // Pass control to the next handler
  },
  (req, res) => {
    res.send('Handler 2 completed!');
  }
);
```

Router Object for Modular Routing

Using the Router object in Express, you can create modular and reusable route handlers.

```
const express = require('express');
const router = express.Router();

// Define routes
router.get('/', (req, res) => {
  res.send('Welcome to the blog!');
});

router.get('/:id', (req, res) => {
  res.send(`Viewing blog post with ID: ${req.params.id}`);
});

// Use the router in the main app
```

```
app.use('/blog', router);
```

- Visiting /blog will display Welcome to the blog!.
- Visiting /blog/123 will display Viewing blog post with ID: 123.

HTTP Methods in Routing

Express supports all HTTP methods. You can define routes for different methods.

```
app.get('/example'  (req  res) => res.send('GET request'));
app.post('/example'  (req  res) => res.send('POST request'));
app.put('/example'  (req  res) => res.send('PUT request'));
app.delete('/example'  (req  res) => res.send('DELETE request'));
```

Handling 404 Errors (Not Found)

If no route matches a request, you can define a catch-all route.

```
app.use((req, res) => {
  res.status(404).send('Page not found!');
});
```

Combining Routes and Middleware

Routing in Node.js is powerful and flexible, enabling clean separation of concerns by combining routes, middleware, and modularity. Use Express.js for a streamlined experience in building robust web applications.

Questions

Why do you think you are the right fit for this Node.js role?

As a Node.js developer, I have experience in building scalable and efficient server-side applications using Node.js. I am a team player and have excellent communication skills. I believe that my experience and skills make me a strong candidate for this Node.js role.

Do you have any past Node.js work experience?

Yes, my past Node.js work experience has given me a solid foundation in building scalable and efficient server-side applications using Node.js.

Do you have any experience working in the same industry as ours?

Yes, I have worked on several Node.js projects in the past.

Do you have any certification to boost your candidature for this Node.js role?

Yes, I am OpenJS Node. Js Services Developer (JSNSD) Certified.