Node Js

Important Note for Students:
This list of questions and answers is like a helpful guide for your upcoming interview. It's designed to give you an idea of what to expect and help you get ready.

But remember:

1. **Variety of Questions:** The same questions can be asked in many different ways, so don't just memorise the answers. Try to understand the concept behind each one.
2. **Expect Surprises:** There might be questions during your interview that are not on this list. It's always good to be prepared for a few surprises.
3. **Use This as a Starting Point:** Think of this material as a starting point. It shows the kind of questions you might encounter, but it's always good to study beyond this list during your course.

---

1. What is Node.js?
Node.js is an open-source JavaScript runtime built on the Chrome V8 JavaScript engine. It allows developers to execute JavaScript code on the server-side, enabling them to build scalable, efficient, and real-time applications.

2. How does Node.js handle asynchronous operations?
Node.js uses an event-driven, non-blocking I/O model. Asynchronous operations are managed using callbacks, promises, or async/await, allowing the server to handle multiple requests concurrently without blocking the execution thread.

3. What is the Node.js package manager?
The Node.js package manager is called npm (Node Package Manager). It's used to install, manage, and share packages (libraries) that extend Node.js functionality.

4. How can you include external modules in Node.js?
You can include external modules using the require() function. For example, to include the express module: const express = require('express');

5. What is the purpose of the fs module in Node.js?
The fs module, short for "file system," provides methods for interacting with the file system, such as reading and writing files, creating directories, and managing paths.

6. What is the difference between require() and import in Node.js?
require() is the CommonJS way of importing modules in Node.js, whereas import is part of the ECMAScript module system introduced in modern JavaScript (ES6+). Node.js supports both, but import requires certain configuration and might not work in all cases without transpilation.

7. How can you create a simple HTTP server using Node.js?
You can create a simple HTTP server using the built-in http module in Node.js. Here's a basic

Example:

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

const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello, Node.js!');
});

server.listen(3000, () => {
  console.log('Server is listening on port 3000');
});
```

8. What is the purpose of the global object process in Node.js?
The process object provides information and control over the Node.js process running on the system. It contains properties like argv (command-line arguments), env (environment variables), and methods like exit() to terminate the process.

9. How can you handle errors in Node.js?
Errors in Node.js can be handled using try...catch blocks, or by attaching error event listeners to asynchronous operations. Additionally, Node.js provides the process.on('uncaughtException') event to catch unhandled exceptions.

10. What is middleware in the context of Express.js?
Middleware in Express.js are functions that are executed in the request-response cycle. They can modify the request and response objects, handle authentication, logging, and other common tasks. Middleware can be added globally or to specific routes using app.use().

11. What is npm and how is it used?
npm (Node Package Manager) is the default package manager for Node.js. It's used to install, manage, and share JavaScript packages (libraries) from the npm registry. You can use commands like npm install, npm uninstall, and npm update to manage packages in your Node.js projects.

12. How can you handle routes and URLs in Node.js using Express.js?
Express.js provides a routing system that allows you to define routes for different URLs and HTTP methods. You can use the app.get(), app.post(), app.put(), and app.delete() methods to define routes and their corresponding handlers.

13. What is middleware in the context of Express.js?
Middleware in Express.js are functions that execute during the request-response cycle. They can perform tasks like logging, authentication, data parsing, and more. Middleware can be added globally or to specific routes using the app.use() method.

14. What is the purpose of the next() function in Express.js middleware?
In Express.js middleware, the next() function is used to pass control to the next middleware in the stack. If not called, the request-response cycle may get stuck. It's crucial to call next() to ensure the middleware chain continues.

15. How can you read environment variables in Node.js?
You can access environment variables using the process.env object in Node.js. For example, to read the value of an environment variable named DATABASE_URL, you would use process.env.DATABASE_URL.

16. How does Node.js handle modules and dependencies?
Node.js uses the CommonJS module system to manage modules. Each file is treated as a separate module. You can use the require() function to import modules, and you can create your own modules using the module.exports object.

17. What is the purpose of the Buffer class in Node.js?
The Buffer class is used to work with binary data in Node.js, including reading from or writing to streams and handling network protocols. It's particularly useful for dealing with file I/O and binary data manipulation.

18. How can you perform asynchronous operations in a loop in Node.js?
To perform asynchronous operations in a loop, you can use techniques like callbacks, promises, or async/await. Promises and async/await are especially helpful for writing more readable and maintainable code when dealing with async loops.

19. What is the event loop in Node.js?
The event loop in Node.js is responsible for managing the execution of asynchronous code. It constantly checks the call stack and the message queue, moving tasks from the queue to the stack when the stack is empty, ensuring non-blocking behavior.

20. What is a module bundler, and why might you use one in Node.js?
A module bundler like Webpack or Parcel is used to bundle JavaScript modules and their dependencies into a single file (or multiple files) for deployment. It's useful for optimizing performance by reducing the number of network requests and managing assets.

21. What is the purpose of the require.resolve() method in Node.js?
The require.resolve() method is used to determine the full path of a module without actually loading the module into memory. It's commonly used when you need to know the location of a module's file.

22. How can you handle file uploads in Node.js with Express.js?
To handle file uploads in Express.js, you can use middleware like multer. Multer simplifies handling multipart/form-data, which is typically used for file uploads. It allows you to access uploaded files and fields in your route handlers.

23. What is a Promise in Node.js?
A Promise is a built-in object in Node.js that represents the eventual completion or failure of an asynchronous operation. It simplifies handling asynchronous code by providing methods to handle success and failure scenarios.

24. What is the async keyword in JavaScript functions?
The async keyword is used to define an asynchronous function in JavaScript. An asynchronous function always returns a Promise, and you can use the await keyword within it to pause execution until a Promise is resolved.

25. What is the purpose of the child_process module in Node.js?
The child_process module in Node.js is used to create and manage child processes. It allows you to execute commands or other Node.js scripts as separate processes, enabling parallel execution and interaction with external programs.

26. How can you handle cross-origin resource sharing (CORS) in Node.js?
To handle CORS in Node.js, you can use middleware like cors to configure your Express.js application to allow or restrict cross-origin requests. This helps control which origins are permitted to access your resources.

27. What is the difference between process.nextTick() and setTimeout()?
process.nextTick() schedules a callback to be executed on the next iteration of the event loop, immediately after the current operation completes. setTimeout() schedules a callback to be executed after a specified delay, allowing other operations to run in the meantime.

28. How can you deploy a Node.js application to a server?
You can deploy a Node.js application by:

Setting up a server environment (e.g., using a cloud service like AWS, Heroku, or DigitalOcean).
Uploading your application files to the server.
Installing Node.js and the necessary dependencies on the server.
Running your application using a process manager like PM2 or systemd.

29. What is the purpose of the os module in Node.js?
The os module in Node.js provides methods to interact with the operating system. It offers information about the server's CPU, memory, network interfaces, and more.

30. What is an EventEmitter in Node.js?
An EventEmitter is a built-in class in Node.js that allows you to create objects that emit named events. You can attach event listeners to these objects to respond to specific events when they occur.

31. How can you read and write files asynchronously in Node.js?
Node.js provides the fs (file system) module for reading and writing files asynchronously. You can use methods like fs.readFile() and fs.writeFile() to perform file operations without blocking the event loop.

32. What is the purpose of the url module in Node.js?
The url module in Node.js provides utilities for working with URLs. It allows you to parse URLs, extract components like hostname and pathname, and create URL objects.

33. How do you handle environment-specific configuration in a Node.js application?
Environment-specific configuration can be managed using environment variables. For example, you can use the process.env object to access environment variables like database credentials, API keys, and configuration settings.

34. What is the role of a reverse proxy in Node.js deployments?
A reverse proxy acts as an intermediary server that forwards client requests to the appropriate backend server (Node.js application). It can enhance security, load balancing, and provide additional features like caching and SSL termination.

35. How can you handle sessions and cookies in a Node.js application?
You can use middleware like express-session in Express.js to manage sessions and cookies. This middleware handles session data and can store it in memory, on disk, or in a database.

36. What is the purpose of the cluster module in Node.js?
The cluster module allows you to create multiple child processes (workers) for a Node.js application. This helps utilize multiple CPU cores, improving performance and allowing better handling of concurrent requests.

37. How can you handle real-time communication in Node.js applications?
For real-time communication, you can use libraries like socket.io that provide WebSocket-based communication between the server and clients. WebSocket enables bi-directional communication for features like real-time chats and live updates.

38. What is the process.argv property in Node.js?
The process.argv property is an array containing command-line arguments passed to a Node.js script. The first element is the path to the Node.js executable, the second element is the script file's path, and subsequent elements are arguments passed when running the script.

39. How can you securely store sensitive information like API keys in a Node.js application?
You can use environment variables to store sensitive information outside of your codebase. Libraries like dotenv can help manage environment variables in development environments. In production, use the environment provided by your hosting platform.

40. What is the purpose of the crypto module in Node.js?
The crypto module provides cryptographic functionality for generating secure hashes, encryption, decryption, digital signatures, and more. It's often used to enhance the security of data in Node.js applications.

41. How can you handle asynchronous errors in Node.js?
Asynchronous errors can be handled using the try...catch construct within asynchronous functions. Additionally, using error-first callbacks, promises with .catch(), and try...catch with async/await can help manage errors effectively.

42. What is a stream in Node.js?
A stream is a way to handle reading from or writing to data sources in a continuous manner, piece by piece, rather than loading the entire data into memory. Streams are crucial for efficient I/O operations, especially when dealing with large files or network data.

43. How do you manage database operations in Node.js applications?
Node.js can interact with databases using various libraries like mongoose (for MongoDB), sequelize (for SQL databases), and pg-promise (for PostgreSQL). These libraries provide abstractions and methods for querying, inserting, updating, and deleting data in databases.

44. What is middleware in the context of Express.js routing?
Middleware in Express.js can be thought of as functions that sit between the initial request and the final response. They can perform tasks like logging, authentication, and validation before passing the request to the route handler.

45. How can you improve the performance of a Node.js application?
Performance improvements can include optimizing database queries, using caching mechanisms (e.g., Redis), implementing load balancing with multiple instances, using a reverse proxy, minifying code, and enabling compression.

46. What is the purpose of the http module in Node.js?
The http module in Node.js provides a way to create and manage HTTP servers and clients. It enables you to create web servers, handle requests, and send responses using HTTP methods like GET, POST, PUT, and DELETE.

47. How can you handle authentication in Node.js applications?
Authentication can be handled using libraries like passport or jsonwebtoken. These libraries help manage user authentication and session management, enabling secure access to resources.

48. What are template engines in Node.js?
Template engines are used to generate dynamic HTML content by combining data with template files. Popular template engines for Node.js include EJS, Handlebars, and Pug (formerly known as Jade).

49. How can you schedule tasks to run at specific intervals in Node.js?
You can use the built-in setInterval() function to repeatedly execute a function at specified intervals. For more advanced scheduling, you can use libraries like node-schedule or node-cron.

50. What is the purpose of the util module in Node.js?
The util module provides utility functions that are helpful for working with JavaScript objects and functions. It includes functions for inheritance, debugging, and formatting.

51. What is a template literal in Node.js?
A template literal is a string literal that supports embedded expressions using backticks (`). It allows you to include variables and expressions directly within the string.

52. What is the purpose of the assert module in Node.js?
The assert module provides assertion testing to verify that values meet expected conditions. It's often used for writing unit tests to ensure code correctness.

53. How can you handle promises that need to be resolved or rejected in a specific order?
You can use Promise.all() to handle multiple promises concurrently and wait for all of them to resolve or reject. For sequential execution, you can use chaining or async/await.

54. What is the Node.js event loop and how does it work?
The Node.js event loop is a fundamental part of the runtime that handles asynchronous operations. It continually processes events from the event queue, executing callbacks and resolving promises when their conditions are met.

55. How can you secure a Node.js application against common vulnerabilities?
To secure a Node.js application, you should:

- Regularly update dependencies to patch security vulnerabilities.
- Implement input validation to prevent injections and XSS attacks.
- Use libraries like helmet to set security-related HTTP headers.
- Use authentication and authorization mechanisms for user access.
- Validate and sanitize user inputs to prevent malicious actions.

56. How does error handling in async/await differ from using callbacks or promises?
In async/await, error handling resembles synchronous code with try...catch blocks. Errors thrown within an async function can be caught using surrounding try...catch blocks.

57. What are WebSocket connections, and how can you implement them in Node.js?
WebSocket connections provide full-duplex communication between a client and a server over a single connection. Libraries like ws allow you to create WebSocket servers and clients in Node.js for real-time bidirectional communication.

58. How can you manage environment-specific configurations using .env files?
Libraries like dotenv allow you to store environment-specific configuration variables in a .env file. These variables can be loaded using process.env to keep sensitive information separate from your codebase.

59. How can you profile and debug Node.js applications?
You can use built-in tools like the Node.js Debugger (node inspect or node --inspect) for debugging. Profiling can be done using the --prof flag with node and analyzing the generated .cpuprofile or .heapprofile files.

60. How do you handle memory leaks in long-running Node.js applications?
To prevent memory leaks, you should:

- Use tools like the --inspect flag or Node.js built-in heapdump to diagnose leaks.
- Avoid global variables and circular references.
- Unsubscribe from event listeners when they're no longer needed.

- Release resources and close connections properly.
- Use heap snapshots and monitoring tools to detect trends over time.

61. What is Authentication?
Authentication is the process of identifying someone's identity by assuring that the person is the same as what he is claiming for. (**Identification**) It is used by both servers and clients.

- Client ⇒ The client uses it when he wants to know that it is the same server that it claims to be.
- Server ⇒
  - The server uses it when someone wants to access the information, and the server needs to know who is accessing the information.
  - It is done mostly by using the *username and password.*
  - Other ways of authentication can be done using *cards, retina scans, voice recognition, and fingerprints.*
- Most common Authentication used
  - Signup/Register/Create Account
  - Login.
- If the Email and Password are correct only then Login.
- We have to match it with the data that is there in the database.
- For Authentication, we give them an access token that is unique when they are logged in.For that, we use JWT(JSON Web Token) Authentication.

62. What is Authorization?
Authorization is the process of granting someone to do something. It means it is a way to check if the user has permission to use a resource or not.

- It usually works with authentication so that the system could know who is accessing the information.
- In authorization, data is provided through the access tokens.
- Authorization permissions cannot be changed by the user. The permissions are given to a user by the owner/manager of the system, and he can only change it.

Example ⇒

- The author only can delete their own post as it is authorized to do that.
- After customers successfully authenticate themselves, they can access the cart page as they have the token with them.

63. How do you do role-based authentication?

- Role-based authentication is a common technique used to manage access control in systems with multiple users. In this approach, each user is assigned a specific role or set of roles that determine their level of access to various resources within the system.
- Identify the different roles that will be needed to access different resources within the system. For example, a system might have roles such as "teachers", "students", or “admin”.
- We can separate out the parts of our application based on features (like roles, such as students, and teachers)with the help of routes.In the above example, separate routes are made for teachers and students.
- We can implement access control with the use of a combination of authentication and authorization mechanisms to enforce access control based on the user's role. This might include using passwords, tokens, or other authentication methods, as well as implementing rules that restrict access based on the user's role and permissions.
- The goal of role-based authentication is to provide a secure and efficient way of managing access to resources within a system, while also ensuring that users are only given access to the resources that they need to perform their assigned tasks.

64. What is hashing?

- Hashing is a process of transforming plain text into a fixed-length sequence of characters, known as a **hash.**
- Hashing is used to store passwords securely in a database.
- When a user creates a password, the password is hashed before it is stored in the database. This ensures that even if the database is compromised, an attacker would not be able to easily obtain the original passwords.
- When the user logs in again, their password is hashed again and compared to the stored hash to verify their identity.
- Hashing algorithms are one-way designed, meaning that it is difficult (or infeasible) to decrypt the original cleartext from the hash. This makes it difficult for attackers to obtain the original password, even if they have access to the hash.
- The **bcrypt** module (**third-party package available for Node.js**) is there that provides an implementation of the bcrypt hashing function. It can be used to securely store passwords in a way that makes them difficult to crack.
- The **bcrypt** module is built on top of the bcrypt algorithm and provides a simple interface for hashing passwords and comparing them to hashed values.
- Hashing provides an extra layer of security, it is not foolproof. Attackers can still use methods such as brute force attacks to try to guess passwords, even if they only have access to the hashed values.
- To overcome this **salt (a random sequence of characters)** is added to the password before it is hashed.

65. What is encryption?

- Encryption is the process of converting plain, readable data or information into an encoded form so that it can only be accessed and understood by authorized parties who have the decryption key.
- The purpose of encryption is to protect data from unauthorized access, theft, or modification.
- The coded message is called **cipher text.**
- The transformation from plain text to cipher text is done using an encryption algorithm and a secret key.
- Decryption (the process of converting encrypted data back into its original form) is required so that it can be read and understood by an authorized recipient.
- For decrypting the cipher text, you require a key or password that was used to encrypt the data, without which the data cannot be decrypted.
- Data like **videos, audios** and even **WhatsApp messages** are sent in encrypted form.

66. How are hashing and encryption different?

HASHING

- Hashing is a one-way process that takes data and produces a fixed-size output, called a **hash**.
- The hash cannot be reversed back to the original data.
- Hashing is used for storing passwords securely, as the hash of a password can be stored instead of the password itself.
- When a user enters their password, the system takes the **hash** of the password and compares it with the stored **hash** to see if they match.

ENCRYPTION

- Encryption is a two-way process that transforms plain text into ciphertext using an encryption algorithm and a key.
- The cipher text can be decrypted to the original data as well.
- Encryption is used for securing data during transmission, such as when sending sensitive information over the internet. such as videos, audio, and even WhatsApp messages.
- The ciphertext can be decrypted back into the plain text using a decryption algorithm and the same key.

67. What is salt?

- **Salt is a random sequence of characters that** is added to the password before it is hashed.

- The **purpose of the salt** is to add an extra layer of security to the password hash.
- When a user creates a password, the password is first combined with a random salt value. The resulting combination is then hashed and stored in the database along with the salt value.
- When the user logs in again, the salt value is retrieved from the database and combined with the entered password. The resulting combination is then hashed and compared to the stored hash value to verify the user's identity.
- A salt value can be generated using a variety of methods :
    - By using a random number generator, or
    - By using unique values such as the user's username or email address.
- It is important to use a unique salt value for each password because using the same salt value for multiple passwords would make the hashes vulnerable to attacks such as "collision attacks".
- Some popular hashing algorithms, such as **bcrypt**, automatically generate a random salt value for each password by default. This helps ensure that the salt values are unique and unpredictable and increases the security of the password hashes.

68. What is JWT?

- **JWT** stands for **JSON Web Token**. It is a compact, URL-safe means of representing claims to be transferred between two parties. JWTs are used for authentication and authorization in web applications and APIs.
- They contain JSON objects which have the information that needs to be shared. Each JWT is also signed using hashing to ensure that the JSON contents (also known as JWT claims) cannot be altered by the client or a malicious party.
- A JWT is composed of three parts:

    - **Header**: Consists of two parts:
        - The signing algorithm that’s being used.
        - The type of token, which, in this case, is mostly “JWT”.

    - **Payload**: The payload contains the claims or the JSON object.
    - **Signature**: A string that is generated via a cryptographic algorithm (hashing) that can be used to verify the integrity of the JSON payload.

69. How is JWT different and list the pros and cons of using JWT tokens?

- JWT is a format for securely transmitting data between parties, while hashing and encryption are cryptographic techniques for transforming data. JWT is used for authentication and authorization in web applications, while hashing and encryption are used for data validation and data confidentiality.

- **PROS of JWT -**

    - **Stateless and Scalable**: JWTs do not require a database or server-side storage to maintain state information. This makes them easier to manage and more scalable.
    - **Security:** JWTs can be digitally signed and encrypted, providing a high level of security for transmitting sensitive information.
    - **Standardized:** JWTs are an open standard, supported by many programming languages and web frameworks, making them widely adopted and easy to integrate into different systems.
    - **Versatile:** JWTs can be used for authentication and authorization purposes, as well as for storing custom user attributes or other metadata.

- **CONS of JWT -**

    - **Payload Size:** JWT tokens can be larger in size than other types of tokens, such as session cookies, which can impact network performance.
    - **No Revocation:** Once a JWT token is issued, it cannot be revoked until it expires, which can be a security risk in certain cases.
    - **Token Validation:** Validating the signature of a JWT token requires additional computation, which can impact performance in high-traffic systems.
    - **Security Risks:** If the encryption or signature algorithms used to generate the JWT token are compromised, the token and the data it contains may be vulnerable to attack.

70. What are the different ways to manage authentication?

There are several ways to manage authentication.

- **MFA(Multi-factor Authentication) -**

    - It requires at least two methods of authentication. This is becoming more common because passwords alone are no longer considered secure.

    - With multi-factor authentication, a user is asked to enter a username and password. If it matches, a code is sent to an authentication app, phone number, email, or another resource then only the user is supposed to have access. The user will then enter that code on the login page.

- **2FA(Two-factor authentication) -**

    - It requires exactly two methods of verification, which would mean entering the user's password in addition to one of the methods (e.g., verifying a code sent to their phone, email, app, etc.).
    - Two-factor authentication is becoming more common with consumer applications because it offers much more security than a password alone without causing a lot of inconvenience for the user.

- **Third-party authentication providers -**

    - Websites can use third-party authentication providers, such as Google, Facebook, or Twitter, to authenticate users. This is known as social login.

- **SSO(Single-sign-on) -**

    - Websites can implement SSO using protocols such as SAML (**Security Assertion Markup Language**) or OAuth to allow users to authenticate once and gain access to multiple websites or applications without having to enter their credentials repeatedly.
    - Within an enterprise setting, using an SSO provider means that employees need to only log in once each day. From there, authentication takes place behind the scenes as the identity provider exchanges verification keys with all of the websites and apps that you have set up to use SSO.

71. What is cookie-based auth?

- **Cookie-based auth** is a method of managing user authentication in web applications where a cookie (**pieces of data used to identify the user and their preferences**) is used to store user authentication information.
- The browser returns the cookie to the server every time the page is requested. Specific cookies like HTTP cookies are used to perform cookie-based authentication to maintain the session for each user.
- When a user logs in to a website, the server generates a unique session identifier and sends it back to the user's browser in the form of a cookie. The browser then stores the cookie, and sends it back to the server with every subsequent request, allowing the server to identify and authenticate the user.
- The cookie usually contains an encrypted or hashed version of the user's authentication credentials, such as a username and password. The server verifies the credentials stored in the cookie and uses them to authenticate the user for subsequent requests.

- Cookies can be set to expire after a certain period of time, or when the user logs out of the website. This helps to ensure that the user's session remains secure and that their authentication information is not compromised.
- Its drawback is that cookies can be intercepted and manipulated by attackers, potentially leading to security vulnerabilities.

72. What is session management?

- Session management is the process of securely managing and maintaining the state of a user's interaction with a web application or system.
- The purpose of session management is to keep track of a user's actions across multiple pages or requests and to persist the data for the duration of the user's session.
- This is typically achieved by assigning a unique session ID to the user and storing relevant information, such as the user's preferences or shopping cart contents, on the server. The session ID is then passed back and forth between the client and the server to ensure that the correct information is retrieved and updated for each request.
- The ultimate goal of session management is to provide a seamless and personalized experience for the user and also ensure the security and reliability of the data being stored.
- The process of retrieving personalized information using a session ID:
    - The web browser (client) sends a request to the server for a web page.
    - The server generates a unique session ID and assigns it to the user. This ID is returned back to the client in the form of a cookie or URL parameter.
    - On later requests, the client sends the session ID back to the server as part of the request.
    - The server uses the session ID to look up the user information from a database or other storage mechanism.
    - The server retrieves the user's personalized information based on the session ID and integrates it into the response. This could include things like a custom greeting, the contents of a shopping cart, or any other information that the user has provided or interacted with during their session.
    - The server returns the personalized response to the client, which displays the web page with the relevant information.
    - The client continues to send requests with the session ID, allowing the server to maintain the state of the user's session and retrieve the appropriate information for each request.
    - When a user **logs out of a web application, the session got a timeout, destruction of all cookies, including the session ID cookie or session ID got deleted**, then their session is typically invalidated. This is done to ensure that the user's information is no longer accessible and to prevent unauthorized access to their account.

73. What is OAuth?

- **OAuth (Open Authorization)** is an open standard and authorization protocol that allows third-party applications to access user data from a resource server (such as a social media platform like Facebook, Gmail, etc., or cloud storage service) on behalf of the user, without requiring the user to share their login credentials with the third-party application.
- The OAuth process involves the client (**the third-party application that wants to access the user's data**) obtaining an access token from the resource server (**the server that hosts the user's data and is responsible for authenticating the user**)after the user authenticates themselves on the resource server. This access token is then used by the client to access the user's data from the resource server. The access token is typically valid for a limited time period and can be revoked by the resource owner (**the user who owns the data that the third-party application wants to access**) at any time.

Express.js + Mongoose

74. How does express work?

- Express is just a framework, that can help us in creating the server in a very easy way.
- Four stages :
    - Creating an express application
    - Creating a new route
    - Starting an HTTP server on a given port number
    - Handling a request once it comes in

Every call to express creates an **app** for you. The app is nothing but a server.

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

- Every **app** instance has standard HTTP **verbs** on it. These methods accept 2 arguments:
    1. first is a **route**, of type string.
    2. second is the **generic callback** that we used with **http.**

so you can create routing like this:

```
app.get("/", (req, res) => {
  //  handle request
});
app.post("/new", (req, res) => {
  //
});
```

- You need to start the server on a port.

```
app.listen(8080,()=>{
        console.log("Server started on 8080")
})
```

- Data from the backend or API response you got is in JSON format.
- The biggest advantage is that express has a lot of middleware, even we can create custom one as well to do anything.

75. What are Middlewares?

- Middleware is the layer that exists between the application components, tools, and devices.
- Middlewares are the functions that get executed before the request reaches the route handler and after the response is sent to the client.
- It sets between request and response.
- Because of the middleware, we get the chance to perform modification, updation, or deletion on request and response objects.
- Using middlewares, we can :
  - change the object
  - check for security
  - check for validation
  - calculate the time taken etc.

SYNTAX ⇒

```
 const middleware=(req,res,next)=>{
        // code
        next()
}
```

- **req** and **res** are request and response objects respectively.
- **next** is a function

- **next ( ) Function ⇒**

  - The next ( ) function plays a vital role in applications' request and response cycle.
  - It is a middleware function that runs the next middleware function once it is invoked.
  - In other words, the Next function is invoked if the current middleware function doesn't end the request and response cycle.

- There are express inbuilt middlewares :

- express.json() ⇒ it just basically parses the JSON. If we use it we don't need to parse it manually.
- express.text() ⇒ If we need to parse a text only.

76. What is the MVC framework?

MVC is an acronym for Model-View-Controller. It is a design pattern for software projects.

Model: The model represents the structure of data, the format, and the constraints with which it is stored. It maintains the data of the application. Essentially, it is the database part of the application.

View: View is what is presented to the user. Views utilize the Model and present data in a form that the user wants. A user can also be allowed to make changes to the data presented to the user. They consist of static and dynamic pages which are rendered or sent to the user when the user requests them.

Controller: The controller controls the requests of the user and then generates an appropriate response which is fed to the viewer. Typically, the user interacts with the View, which in turn generates the appropriate request, this request will be handled by a controller. The controller renders the appropriate view with the model data as a response. So, to sum it up:

- Model is data part.
- View is User Interface part.
- Controller is request-response handler.
- **Model** ⇒ The models and all that we created
- **View** ⇒ Frontend/UI (Won't be seeing this)
- **Controller** ⇒ Controlling Logic/Routes

MVC architecture

77. How do you do static routing?

- Static routing is a method of routing network traffic by manually configuring the network devices to use a fixed path or route to reach a particular network or destination.
- With static routing, we must manually provide/configure the above functions on each router of the network. Since all configurations are manual, a routing protocol is not required and is not used in static routing.
- It uses paths between two paths and cannot be automatically updated. Therefore, you must manually reconfigure the static routes when the network changes. It uses low bandwidth compared to dynamic cards. It can be used in areas where network traffic is predictable and designed. It cannot be used in a vast and ever-changing network, because it cannot respond to changes in the network.

78. What are common libraries you work with express?

CORS — Allow or Restrict Requested Resources on a Web Server

- CORS is a node.js package that provides a Connect/Express middleware for enabling CORS with a variety of options.
- CORS stands for Cross-Origin Resource Sharing. Without prior consent, it prevents other websites or domains from accessing your web resources directly from the browser.
- **Features of CORS**
  - Supports GET, POST, or HEAD HTTP methods.
  - Allows web programmers to use regular XML Http Request, which handles errors better.
  - Allows websites to parse responses to increase security.
- **Usage of CORS**
  - We can use either enable CORs for all the routes or only for a single route.

Cookie-parser — Parse Cookies

- Cookie-parser is a middleware that transfers cookies with client requests.
- Cookie-parser uses the req.cookies property to access Cookie data. After parsing, the req.cookies object holds cookies sent by request in JSON format.
- It is capable of parsing both unsigned and signed cookies.
- **Features of Cookie-parser**
  - The decode function is there to decode the value of the cookie.
  - Handle cookie separation and encoding.
  - Can enable signed cookie support by passing a secret string.
  - supports special "JSON cookies" with JSON.parse.

Morgan— Log HTTP Requests and Errors

- Morgan is an HTTP request logger middleware for N typically used for apps.
- It streamlines the process by generating logs for each API request and error. The best fact is that you can utilize a predefined format or design one from scratch, depending on your requirements.
- **Features of Morgan**
  - It Logs the HTTP requests along with some other information. You can also configure what you choose to log.
  - Very helpful in debugging and also if you want to create Log files.

79. How do you manage sessions in express?

- In Express, sessions are typically managed using middleware that stores session data on the server and sends a session ID to the client in a cookie.
- The client then sends the session ID with subsequent requests, allowing the server to retrieve the session data.
- The middleware will create a session object for each client and store it on the server. It will also send a session ID cookie to the client, which will be used to identify the session on subsequent requests.

80. What is CORS?

CORS (Cross-Origin Resource Sharing) —

- CORS stands for Cross-Origin Resource Sharing. Without prior consent, it prevents other websites or domains from accessing your web resources directly from the browser.
- **Features of CORS**
  - Supports GET, POST, or HEAD HTTP methods.
  - Allows web programmers to use regular XML Http Request, which handles errors better.
  - Allows websites to parse responses to increase security.
- **Usage of CORS**
  - We can use either enable CORs for all the routes or only for a single route.

81. How do you manage cookies with express?

- Cookies are small pieces of data that are stored on the client side (usually in the browser) and are sent back to the server with each subsequent request. They are commonly used to store user preferences, session data, and other information that needs to persist across multiple requests.
- To manage cookies with Express, we use the **cookie-parser** middleware. This middleware parses cookies that are sent with the request and makes them available in the **req.cookies** object.
- It is capable of parsing both unsigned and signed cookies.
- **Features of Cookie-parser**
  - The decode function is there to decode the value of the cookie.
  - Handle cookie separation and encoding.
  - Can enable signed cookie support by passing a secret string.
  - supports special "JSON cookies" with JSON.parse.

Mongoose

82. What are Models?

- A constructor is a blueprint using which we can create objects. Similarly, using the Model we create documents (collection of a particular database of MongoDB).
- So, the model is a constructor function.

Syntax ⇒

mongoose.model(NameoftheCollection, SchemaoftheCollection)

- This function returns the **Mongoose object.**

83. What are aggregation pipelines?

- It is a way provided by MongoDB to perform aggregation.
- The aggregation pipeline has various stages and each stage transforms the document.
- It is a multi-stage pipeline. In each stage, the documents are taken as input and produce the resultant set of documents now in the next stage (id is available) the resultant documents are taken as input and produce output, this process is going on till the last stage.
- The basic pipeline stages provide filters that will perform like queries and the document transformation modifies the resultant document and the other pipeline provides tools for grouping and sorting documents.

Example ⇒

Find all the small-size pizzas with prices> 16

db.orders.aggregate([{$match : {size : "small",price : {$gt:16}}}])

$match: It is used for filtering the documents and can reduce the number of documents that are given as input to the next stage

- the **aggregate()** function is used to perform aggregation

84. Explain why a mongoose does not return a promise but has a .then?

- Mongoose provides a way to define and interact with MongoDB collections in a more structured and consistent manner than using MongoDB directly.
- It does not return Promises directly, but it does provide support for Promises through its underlying Promise library, and its fluent API and chaining syntax aim to provide a more consistent way to work with MongoDB collections.

- Mongoose is primarily designed for use in a synchronous, object-oriented programming style, rather than a functional programming style with Promises. By providing a fluent API and supporting synchronous chaining, Mongoose aims to make it easier for developers to work with MongoDB collections in a structured and consistent manner, without having to deal with the complexities of asynchronous programming and Promises.

APIs Questions

85. What is REST API?

- A REST (**Representational State Transfer**) API (**Application Programming Interface**) is a type of web service that enables different software systems to communicate with each other over the internet. RESTful APIs rely on the HTTP protocol, which is built on top of TCP/IP (Transmission Control Protocol/Internet Protocol).
- **REST** suggests creating an object of the data requested by the client and sending the values of the object in response to the user.
- RESTful APIs use **TCP (a low-level networking protocol that provides a reliable, ordered, and error-checked delivery of packets between two applications running on different hosts)** to establish a connection between a client and a server and to transfer data between them.
- When a client makes a request to a server, it sends a TCP packet containing the HTTP request, including the HTTP method (such as GET or POST), the URL, and any parameters or headers.
- The server receives the packet and sends back a TCP packet containing the HTTP response, including the status code, any headers, and the response body.
- A REST API uses HTTP methods, such as GET, POST, PUT, and DELETE, to perform operations on resources. The resources can be represented in different formats, such as XML, JSON, or plain text, depending on the requirements of the client application.
- **Principles of REST API**
  - **Stateless:** A REST API is stateless, which means that each request contains all the necessary information to perform the operation, and the server does not maintain any session state between requests.
  - **Client-server architecture:** A REST API follows a client-server architecture, where the client and server are independent and can evolve separately.
  - **Cacheable:** A REST API can be designed to be cacheable, which means that clients can store responses and reuse them for subsequent requests, which can improve performance.
  - **Uniform interface:** A REST API uses a uniform interface to interact with resources, an HTTP method, and a representation of the resource.

- **Layered system:** A REST API can be designed to be layered, where intermediate servers can act as caches, which can improve scalability and performance.

86. What is gRPC?

- gRPC is a high-performance RPC framework/technology built by Google. It uses Google's own "**Protocol Buffers**", which is an open-source message format for data serialization, as the default method of communication between the client and the server.
- It allows clients to call remote methods on a server as if they were local methods, which makes it easier to build distributed systems and microservices.
- It uses HTTP/2 (**a major revision of the HTTP network protocol used by the World Wide Web**) as the underlying transport protocol, which provides several benefits, such as
  - bi-directional streaming (**clients and servers can send and receive messages at the same time**),
  - flow control, and
  - multiplexing.

87. What is GraphQL?

- GraphQL or "**Graph Query Language**" is a query language and runtime that was developed by Facebook that allows the client (frontend) to request data from an API.
- It provides a more efficient, powerful, and flexible alternative to REST APIs for building web applications.
- Advantages of GraphQL over its RESTful counterpart:
  - **One endpoint:** With traditional REST APIs, you have to create specific endpoints based on the data you want to request. This makes scaling your API difficult because soon, you might find yourself having to manage tens, maybe hundreds, of routes that you will have to remember.
  - **Fewer server requests:** It allows you to make multiple queries and mutations with only one server request. This can be useful when your server only allows a limited number of requests a day.
  - **Declarative data fetching:** It only fetches what you actually need. All you have to do is specify what fields to return.
  - **Type system:** It uses a type system to describe your data, which makes developing much easier. If you are a TypeScript fan, this is a win-win.
  - **Self-documenting:** It is self-documenting, meaning that all of your queries and mutations will automatically be documented by GraphQL.

88. What is HTTP?

- **HTTP** stands for **Hypertext Transfer Protocol**. It's a set of rules which govern the exchange of data over the Internet.

- It is an application-layer protocol that is used to transmit data between web servers and web browsers.
- HTTP defines a set of rules for how web browsers and web servers should communicate with each other. A client (usually a web browser) sends an HTTP request to a server, and the server responds with an HTTP response.
- HTTP also defines several status codes that are used to indicate the status of the request or response. Some common status codes include
  - 200 OK (success).
  - 404 Not Found (resource not found).
  - 500 Internal Server Error (server error).
  - 401 Unauthorized (authentication required).

89. What is a web socket?

WebSocket is a two-way computer communication protocol over a single TCP.Unlike traditional HTTP communication, which requires the client to constantly poll the server for new information.

- It enables real-time communication where the client and server can send data to each other at any time, without constant requests.
- They are commonly used in real-time applications. Examples ⇒
  - chat rooms,
  - online gaming,
  - stock tickers, and
  - other applications
- that require frequent updates and low-latency communication between the client and server.

Node js General Question Answer

90. What is Node.js?

Node.js is a runtime environment that allows developers to run JavaScript code on the server side. It uses the V8 JavaScript engine, the same engine that powers the Google Chrome browser, to execute JavaScript code outside of a browser environment. This enables developers to build scalable and efficient server-side applications using JavaScript.

Example: Node.js allows you to write server-side code in JavaScript, providing a unified language for both front-end and back-end development.

91. How does Node.js work?

Node.js follows an event-driven, non-blocking I/O model. It uses an event loop to handle incoming requests and executes JavaScript code asynchronously. When a request is received, Node.js registers a callback function and continues processing other requests without blocking. Once the I/O operation is complete, the callback function is invoked, allowing the application to respond to the request.

Example: When a web server built with Node.js receives a request for a webpage, it can continue processing other requests while fetching data from a database or making API calls. Once the data is ready, the callback function is executed to send the response back to the client.

92. What are the advantages of using Node.js?

Node.js offers several advantages, including:

- Asynchronous and non-blocking: Node.js handles multiple requests efficiently without blocking the execution of other tasks, resulting in high scalability and performance.

Single-threaded event loop:

- Node.js can handle thousands of concurrent connections with a single thread, reducing memory consumption and enabling real-time applications.

JavaScript ecosystem:

- Node.js leverages the extensive JavaScript ecosystem, allowing developers to reuse libraries and frameworks, both on the front-end and back-end.

Fast development cycle:

- Node.js provides a streamlined development experience with features like npm (Node Package Manager), which offers a vast collection of open-source modules.

Example: Node.js can be beneficial for building real-time chat applications, streaming platforms, or APIs that require high concurrency and low latency.

93. Explain the concept of non-blocking I/O in Node.js.

In Node.js, non-blocking I/O refers to the ability to perform I/O operations asynchronously without blocking the execution of other tasks. Instead of waiting for an I/O operation (such as reading from a file or making a network request) to complete, Node.js registers a callback function and continues executing other code. Once the I/O operation is finished, the callback is invoked, allowing the program to handle the result.

Example: When a Node.js application needs to read a file, it initiates the operation and proceeds to execute other code. Once the file is read, the callback function is called, and the application can process the file's content or perform additional operations.

94. What is an event-driven programming paradigm?

Event-driven programming is a programming paradigm where the flow of the program is determined by events. An event is a signal or notification that something has occurred, such as a button click, a network request, or a file being written. Instead of following a sequential flow, event-driven programs respond to events by executing predefined event handlers or callbacks.

Example: In a Node.js application, a server can listen for incoming HTTP requests as events. When a request arrives, the server triggers the associated event handler, allowing the application to process the request and generate a response.

95. What is the role of the EventEmitter class in Node.js?

The EventEmitter class is a core module in Node.js that allows objects to emit named events and register listeners to those events. It serves as the foundation for implementing event-driven programming in Node.js. Objects that inherit from the EventEmitter class can emit events, and other objects can listen to those events and execute appropriate event handlers when the events occur.

Example: In a Node.js application, the EventEmitter class enables communication between different modules or components. For instance, an event emitter object can emit a "dataReceived" event, and another object can listen to that event and perform actions when new data is received.

96. How can you create a server in Node.js?

In Node.js, you can create a server using the built-in `http` module. Here's an example of how to create an HTTP server that listens for incoming requests:

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

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello, world!');
});

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

This code sets up an HTTP server that listens on port 3000 and responds with "Hello, world!" for all requests. When a request is received, the provided callback function is executed, allowing you to handle the request and send a response.

97. How do you handle asynchronous operations in Node.js?

In Node.js, you can handle asynchronous operations using callbacks, promises, or async/await syntax.

Example with callbacks:

```javascript
fs.readFile('file.txt', 'utf8', (err, data) => {
  if (err) {
        console.error('Error reading file:', err);
```

```
  } else {
        console.log('File contents:', data);
  }
});
```

Example with promises:
```javascript
const readFile = require('fs').promises.readFile;

readFile('file.txt', 'utf8')
  .then((data) => {
        console.log('File contents:', data);
  })
  .catch((err) => {
        console.error('Error reading file:', err);
  });
```

Example with async/await:
```javascript
async function readFileContents() {
  try {
        const data = await fs.promises.readFile('file.txt', 'utf8');
        console.log('File contents:', data);
  } catch (err) {
        console.error('Error reading file:', err);
  }
}

readFileContents();
```

98. What is the purpose of the package.json file?

The package.json file is a manifest file in Node.js projects that contains metadata and configuration information about the project. It serves multiple purposes, including:

- Defining project dependencies: The package.json file specifies the required dependencies (external libraries or modules) for the project. This allows others to install the dependencies easily and ensures consistent development environments.

Executing scripts:

- The file can define scripts that automate various tasks, such as running tests, building the project, or starting the application.

Storing project metadata:

- The package.json file includes information about the project, such as its name, version, author, license, and repository.

Example of a simple package.json file:
```json
{
  "name": "my-project",
  "version": "1.0.0",
  "description": "A description of my project",
  "author": "Your Name",
  "license": "MIT",
  "dependencies": {
        "express": "^4.17.1",
        "lodash": "^4.17.21"
  },
  "scripts": {
        "start": "node index.js",
        "test": "mocha"
  }
}
```

99. How do you manage dependencies in Node.js?
Dependencies in Node.js can be managed using npm (Node Package Manager), which is a command-line tool and an online registry of JavaScript packages. To manage dependencies in a Node.js project:

- Create a package.json file by running `npm init` in your project directory or using `npm init -y` for a default setup.
- Add dependencies to your project by running `npm install <package-name>` for each package you need. This will download the package and update the package.json file with the dependency information.
- The dependencies will be stored in the "dependencies" section of the package.json file, along with their versions.
- To install all the dependencies listed in the package.json file, you can run `npm install` in your project directory.
- When sharing your project with others, they can install the dependencies by running `npm install` based on the package.json file.

Example:
```json
{
  "name": "my-project",
  "version": "1.0.0",
  "dependencies": {
        "express": "^4.17.1",
        "lodash": "^4.17.21"
  }
}
```

100. What is the difference between `require()` and `import` in Node.js?
In Node.js, `require()` is the traditional way of including modules, whereas `import` is part of the ES modules syntax introduced in ECMAScript 6 (ES6).

`require()`:
- Used in CommonJS modules (Node.js default module system).
- Supports dynamic loading and can load modules conditionally at runtime.
- Uses synchronous loading, meaning the execution is blocked until the required module is loaded.
- The imported module is assigned to a variable.

Example:
```javascript
const express = require('express');
const myModule = require('./myModule');
```

`import`:
- Used in ECMAScript modules (ES modules).
- Supports static loading, and all imports must be at the top level of the file.
- Supports asynchronous loading with dynamic imports.
- The imported module can be selectively imported with destructuring.

Example:
```javascript
import express from 'express';
import { someFunction } from './myModule';
```

It's important to note that while Node.js supports `import` for ECMAScript modules, it's not fully supported in all versions or enabled by default. The use of `import` requires additional configuration or the use of transpilers like Babel.

101. How can you handle errors in Node.js?

In Node.js, errors can be handled using try-catch blocks for synchronous code and using error-first callbacks, promises, or async/await for asynchronous code.

Example with try-catch (synchronous code):
```javascript
try {
  // Synchronous code that may throw an error
  const result = someSyncFunction();
  console.log('Success:', result);
} catch (error) {
  console.error('Error:', error);
}
```

Example with error-first callbacks (asynchronous code):
```javascript
asyncFunction((error, result) => {
  if (error) {
        console.error('Error:', error);
  } else {
        console.log('Success:', result);
  }
});
```

Example with promises (asynchronous code):
```javascript
someAsyncFunction()
  .then((result) => {
        console.log('Success:', result);
  })
  .catch((error) => {
        console.error('Error:', error);
  });
```

Example with async/await (asynchronous code):
```javascript
async function someAsyncFunction() {
  try {
        const result = await anotherAsyncFunction();
        console.log('Success:', result);
  } catch (error) {
        console.error('Error:', error);
  }
}
```

102. What is middleware in the context of Node.js?

In Node.js, middleware refers to functions or code snippets that are executed between the initial processing of a request and the final processing of a response in an application's request-response cycle. Middleware functions have access to the request and response objects and can modify them or perform additional operations.

Middleware can be used for various purposes, such as logging, authentication, handling errors, parsing request bodies, or serving static files. It allows developers to modularize the application logic and apply reusable functionality across multiple routes or endpoints.

Example of a simple middleware function in Express.js:
```javascript
function loggerMiddleware(req, res, next) {
  console.log(`${req.method} ${req.url}`);
  next(); // Call the next middleware or route handler
}

app.use(loggerMiddleware);
```

In this example, the `loggerMiddleware` function logs the HTTP method and URL of every incoming request. By calling `next()`, it passes control to the next middleware or route handler in the chain.

103. How can you debug a Node.js application?

Node.js provides various debugging techniques and tools to debug applications, including:

- **Logging:** Adding logging statements throughout the code to output information, errors, and debug messages.
- **Debugging with the built-in debugger:** Node.js includes a built-in debugger that can be enabled by running the application with the `--inspect` flag.
- **Debugging with Chrome DevTools:** Node.js can be debugged using the Chrome DevTools by running the application with the `--inspect` flag and opening the dedicated Chrome DevTools URL.
- **Debugging with IDEs and editors:** Many IDEs and editors, such as Visual Studio Code, provide integrated debugging support for Node.js applications.

Example of debugging with logging:
```javascript
console.log('Starting the application...');

// Debugging code
console.log('Value of variable:', someVariable);

console.log('Processing...');

// Debugging code
console.log('Data received:', responseData);

console.log('Finishing the application...');
```
In this example, logging statements are added to output the state of variables and the flow of the application during execution.

104. Explain the concept of streams in Node.js.
Streams in Node.js provide an abstraction for handling data in chunks rather than loading it all into memory at once. A stream is an object that represents a sequence of data that can be read from or written to.

There are several types of streams in Node.js, including readable streams, writable streams, and duplex streams (both readable and writable). Streams can be used for various purposes, such as reading from or writing to files, network communication, or processing large amounts of data efficiently.

Example of reading a file using streams:
```javascript
const fs = require('fs');

const readStream = fs.createReadStream('largeFile.txt');

readStream.on('data', (chunk) => {
  // Process each chunk of data
  console.log('Received data chunk:', chunk);
});

readStream.on('end', () => {
  console.log('Finished reading the file.');
});
readStream.on('error', (error) => {
  console.error('Error reading the file:', error);
});
```
In this example, a readable stream (`readStream`) is created from a file, and it emits `data` events as it reads the file in chunks. The data can then be processed or written to another stream or destination.

105. How do you handle file operations in Node.js?
Node.js provides the built-in `fs` module for working with the file system. It offers functions for performing various file operations, such as reading from or writing to files, creating directories, or renaming files.

Example of reading from a file using the `fs` module:
```javascript
const fs = require('fs');

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

In this example, the `readFile` function is used to read the contents of the file named `file.txt`. The callback

function receives an error object (`err`) if an error occurs during the file read operation, and the data read from the file is available as a string (`data`) if the read operation is successful.

106. What is Authentication?

Authentication is the process of verifying the identity of a user or system. In the context of web applications, authentication ensures that only authorized users can access certain resources or perform specific actions. It typically involves validating credentials, such as usernames and passwords, to grant or deny access to protected areas of an application.

Example: When a user logs into a website by providing their username and password, the application authenticates the user's credentials to verify their identity. If the credentials are valid, the user is granted access to restricted areas or functionality.

107. What is Authorization?

Authorization is the process of granting or denying access to specific resources or actions based on the authenticated user's privileges or permissions. Once a user is authenticated, authorization determines what the user is allowed to do within the application. It involves defining access control rules and enforcing them to protect sensitive data or functionality.
Example: After a user is authenticated, authorization determines whether they have permission to view, edit, or delete certain resources. For example, a user with an "admin" role may have access to all functionalities, while a regular user may have limited access.

108. How do you do role-based authentication?

Role-based authentication involves associating different roles or groups with users and granting permissions based on those roles. It allows administrators to manage access control at a more granular level by assigning roles to users and defining what each role can or cannot do within the application.

Example: In a web application, you can assign roles like "admin," "moderator," or "user" to different users. Each role may have specific permissions associated with it, such as the ability to create, read, update, or delete certain resources. Role-based authentication ensures that users only have access to the functionalities permitted by their assigned role.

109. What is hashing?

Hashing is the process of converting data of any size into a fixed-size string of characters, called a hash value or hash code. The hash function takes the input data and performs a one-way mathematical operation to generate the hash value. Hashing is commonly used for password storage and data integrity verification.

Example: When storing user passwords in a database, it's considered best practice to hash the passwords instead of storing them in plain text. The password is transformed into a hash value using a cryptographic hash function. When a user enters their password during authentication, it is hashed again, and the hash is compared with the stored hash to verify the password's correctness.

110. What is the purpose of the `util` module in Node.js?
The `util` module in Node.js provides various utility functions that are commonly used in JavaScript programming. It contains functions for working with objects, arrays, strings, and other types, as well as functions for asynchronous control flow, debugging, and formatting.

Example: The `util` module provides the `promisify` function, which converts callback-based functions into functions that return promises. This allows you to work with callback-based APIs using promises or async/await syntax.

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

const readFile = util.promisify(fs.readFile);

readFile('file.txt', 'utf8')
  .then((data) => {
        console.log('File contents:', data);
  })
 .catch((err) => {
        console.error('Error reading file:', err);
  });
```

In this example, the `promisify` function is used to convert the `readFile` function from the `fs` module, which follows the error-first callback style, into a promise-based function.

111. What are the differences between Node.js and browser JavaScript?
While both Node.js and browser JavaScript are based on the JavaScript programming language, there are some key differences between them:

Execution environment: Node.js is a runtime environment that allows JavaScript to be executed outside of a web browser, on the server side. Browser JavaScript runs within web browsers.
APIs and capabilities: Node.js provides APIs for file system operations, network communication, and server-side functionality, which are not available in browser JavaScript. On the other hand, browser JavaScript has APIs for manipulating the Document Object Model (DOM), interacting with web APIs, and handling browser events.

Modules and dependency management: Node.js has a built-in module system that allows modular development and dependency management using `require()` and `npm`. In the browser, modules can be handled using tools like bundlers or module loaders, such as webpack or RequireJS.
Global objects: Node.js has global objects and modules specific to its runtime environment, such as `global`, `process`, and `fs`. Browser JavaScript has its own global objects, such as `window` and `document`.
Concurrency model: Node.js uses an event-driven, non-blocking I/O model, allowing it to handle multiple concurrent connections efficiently. Browser JavaScript follows a single-threaded model and can be blocked by long-running tasks.
Development tools: Node.js has tools and frameworks tailored for server-side development, such as Express.js and Nest.js, while browser JavaScript has tools like the browser's developer console, React, Angular, or Vue.js for client-side development.

112. How can you deploy a Node.js application?
Deploying a Node.js application involves making it available for use on a server or hosting platform. The specific deployment process can vary depending on the hosting environment and requirements of the application. Here are some common steps involved:

Prepare the application: Ensure that the application is ready for deployment by optimizing the code, configuring environment variables, and bundling/minifying assets if necessary.

Choose a hosting platform: Select a hosting platform that suits your requirements, such as cloud providers like AWS, Azure, or Google Cloud, or platform-as-a-service (PaaS) providers like Heroku or Netlify. Each platform may have its own deployment process.

Set up the server or platform: Provision a server or create a project/app on the hosting platform. Configure any necessary settings, such as server configurations, environment variables, or project-specific settings.

Build and package the application: Generate a production-ready build of your Node.js application. This may involve bundling JavaScript files, compiling TypeScript code, or running any necessary build scripts.

Deploy the application: Upload or deploy the application to the hosting platform. This can be done via command-line tools, FTP, Git, or a platform-specific deployment process.

Configure and scale: Configure any necessary settings, such as domain names, SSL certificates, database connections, and scaling options to handle increased traffic or demand.

Test and monitor: Perform testing on the deployed application to ensure it functions as expected. Set up monitoring tools to track the application's performance, errors, and usage.

Continuous deployment (optional): Set up a continuous integration and deployment (CI/CD) pipeline to automate the deployment process whenever changes are made to the codebase. This can involve using tools like Jenkins, Travis CI, or GitLab CI/CD.

113. What are some popular frameworks built on top of Node.js?
Node.js has a rich ecosystem of frameworks and libraries that can be used to build web applications, APIs, and more. Some popular frameworks built on top of Node.js include:

Express.js: A minimal and flexible web application framework that provides robust routing, middleware, and utility features for building web applications and APIs.
Nest.js: A progressive and opinionated framework for building server-side applications, emphasizing developer productivity, code organization, and extensibility.
Hapi.js: A rich framework for building web applications and APIs with a focus on configuration-driven development, code reusability, and plugin architecture.
Sails.js: A full-featured MVC framework for building real-time applications, REST APIs, and microservices with support for data-driven APIs, websockets, and easy integration with front-end frameworks.
Meteor.js: A full-stack framework for building real-time web and mobile applications using a single codebase, with built-in data synchronization, live reloading, and hot code pushes.
Adonis.js: A full-stack web application framework that follows the MVC pattern and provides a robust set of features for building scalable and performant applications.
LoopBack: A highly extensible framework for building APIs and microservices, with built-in support for generating Swagger documentation and easy integration with data sources and databases.
Koa.js:

- A full-featured web application framework that follows the MVC (Model-View-Controller) pattern and provides a wide range of features for building scalable and modular applications.
- A lightweight framework that focuses on simplicity and modularity, providing an elegant way to write middleware-based applications with ES6 syntax and async/await support.

These frameworks provide a higher-level abstraction, routing, middleware, and other features that simplify the development of Node.js applications and APIs. They offer different architectural patterns, levels of complexity, and focus areas, allowing developers to choose the framework that best suits their project requirements and coding preferences.