1. Node.js Basics

1. What is Node.js? Why use it?

Node.js is a **runtime environment** that allows you to execute **JavaScript** code on the server side, outside of a web browser. It's built on the **Chrome V8 JavaScript engine**. Node.js is primarily used for building **scalable, high-performance, and real-time network applications**. It is especially popular for backend services, APIs, and microservices.

2. Difference between Node.js and JavaScript.

JavaScript is a **programming language**, while **Node.js** is a **runtime environment** that lets you run JavaScript code on the server. Think of it this way: you can't drive a car (JavaScript) without an engine (V8) and a chassis (Node.js runtime).

3. What is V8 engine?

The **V8 engine** is a **high-performance open-source JavaScript and WebAssembly engine** developed by Google for Chrome and other web browsers. It compiles JavaScript code into **fast machine code** directly, rather than interpreting it, which is why Node.js is so fast.

4. Difference between Node.js and traditional server-side technologies (Java, PHP).

Traditional server-side technologies like Java and PHP are typically based on a **multi-threaded, blocking I/O model**. This means a new thread is created for each incoming request, which can be resource-intensive. Node.js, on the other hand, uses a **single-threaded, non-blocking I/O model** with an **event loop**. This makes it highly efficient for handling a large number of concurrent connections with minimal overhead.

5. What are advantages & disadvantages of Node.js?

Advantages:

- **High performance** due to its non-blocking I/O and event-driven architecture.
- **Scalability** for real-time applications.
- **Unified language** for both frontend and backend development (JavaScript).
- Large **NPM ecosystem**.

Disadvantages:

- Not suitable for **CPU-intensive tasks** because it can block the single thread.
- **Callback hell** can be a problem without proper handling (e.g., Promises or async/await).

- **Maturity:** some libraries and frameworks are not as mature as those in other languages.

6. What is the event loop in Node.js?

The **event loop** is a core component of Node.js's non-blocking I/O model. It continuously checks the **call stack** and the **task queue**. When the call stack is empty, it moves tasks from the task queue to the call stack for execution. This mechanism allows Node.js to handle asynchronous operations efficiently without creating new threads for each request.

7. What is single-threaded architecture? How does Node.js handle concurrency?

The **single-threaded architecture** means that Node.js uses only one main thread for all requests. However, it achieves **concurrency** through its **non-blocking I/O model** and the **event loop**. While a request is waiting for a slow I/O operation (like a database query or file read), the main thread doesn't sit idle. Instead, it processes other incoming requests. Once the I/O operation is complete, the callback is placed in the task queue and eventually handled by the event loop.

8. What are global objects in Node.js?

Global objects are objects available in all modules without explicitly being imported. Some key global objects include:

- `process`: provides information about the current Node.js process.
- `console`: for printing output to the console.
- `__dirname`: the directory name of the current module.
- `__filename`: the file name of the current module.
- `module`: a reference to the current module.
- `require`: for importing modules.

9. Difference between `require()` and `import`.

`require()` is part of the **CommonJS** module system, which is the traditional way of handling modules in Node.js. It's a **synchronous** function. `import` is part of the **ES6 (ECMAScript 2015)** module system. It's an **asynchronous** function and is the modern standard for JavaScript. The `import` syntax is generally preferred for its static analysis benefits and cleaner syntax.

10. What is REPL in Node.js?

REPL stands for **Read-Eval-Print-Loop**. It's an interactive command-line interface that allows you to execute JavaScript code and see the results instantly. You can access it by typing `node` in your terminal. It's useful for debugging and quick code execution.

---

2. Modules & NPM

11. What are modules in Node.js? Types of modules.

Modules are encapsulated units of code that can be reused in different parts of an application. They help organize code and prevent global namespace pollution. There are three types of modules in Node.js:

- **Core Modules:** Built-in modules that come with Node.js, like `fs` (file system) and `http`.
- **Local Modules:** Modules created by the user for a specific application.
- **Third-party Modules:** Modules installed from **NPM** (Node Package Manager).

12. Difference between CommonJS and ES6 modules.

- **CommonJS** (`require`/`module.exports`):
  - **Synchronous loading**: modules are loaded sequentially.
  - **Dynamic**: `require()` can be called conditionally.
  - Primarily used in Node.js.
- **ES6 Modules** (`import`/`export`):
  - **Asynchronous loading**: can load modules in parallel.
  - **Static**: `import` statements must be at the top level of the file.
  - The standard for modern JavaScript (both browser and Node.js).

13. How to create custom modules in Node.js?

You can create a custom module by defining functions, variables, or objects in a JavaScript file and then exporting them using `module.exports`. For example, in a file named `my-module.js`:

JavaScript

```
// my-module.js
const greet = (name) => {
  return `Hello, ${name}!`;
};

module.exports = {
  greet,
  // You can export multiple things
};
```

Then, you can use it in another file like this:

JavaScript
// app.js
const myModule = require('./my-module.js');
console.log(myModule.greet('Alice')); // Output: Hello, Alice!

14. What is NPM? Difference between `dependencies` and `devDependencies`.

NPM (Node Package Manager) is the default package manager for Node.js. It's a vast registry of open-source packages and a command-line tool for installing, managing, and publishing them.

- **`dependencies`**: Packages required for the **production** operation of your application (e.g., `express`, `mongoose`). They are essential for the app to run.
- **`devDependencies`**: Packages only needed for **development** and testing (e.g., `jest`, `nodemon`). They are not required in the production environment.

15. What is the difference between local and global NPM packages?

- **Local packages** are installed in the `node_modules` folder of your project. They are specific to that project and are version-controlled.
- **Global packages** are installed in a system-wide directory. They are usually command-line tools you want to use from any directory on your machine (e.g., `nodemon`, `create-react-app`).

16. What is `package.json` and `package-lock.json`?

- **`package.json`**: A manifest file that contains metadata about a project, including its name, version, description, and, most importantly, its **dependencies**. It defines the packages your project needs, often with a version range.
- **`package-lock.json`**: A file automatically generated by NPM. It records the **exact versions** of all dependencies (including sub-dependencies) that were installed. This ensures that everyone working on the project has the same dependency tree, preventing version-related bugs.

17. What is semantic versioning (`^`, `~` in package.json)?

Semantic Versioning (SemVer) is a versioning system with three numbers: **MAJOR.MINOR.PATCH**.

- **MAJOR** version is incremented for incompatible API changes.
- **MINOR** version is for new, backward-compatible functionality.
- **PATCH** version is for backward-compatible bug fixes.

In `package.json`:

- `^` (caret): `^1.2.3` means "compatible with version 1.2.3 or greater, but less than 2.0.0." It allows minor and patch updates.
- `~` (tilde): `~1.2.3` means "compatible with version 1.2.3 or greater, but less than 1.3.0." It only allows patch updates.

18. What are peer dependencies in NPM?

Peer dependencies are dependencies that a package needs but does not install itself. Instead, it assumes that the host application will provide them. They are often used by plugins or libraries that integrate with a specific framework. For example, a React component library might list `react` as a peer dependency, expecting the main application to have it installed.

19. What is `npx` and how is it different from `npm`?

- `npm` is a package **installer**. It's used to install, manage, and publish packages.
- `npx` is a package **runner**. It's used to execute an NPM package without having to install it globally. This is useful for running one-off command-line tools.

20. How to update and uninstall NPM packages?

- **Update:**
  - `npm update [package-name]` to update a specific package.
  - `npm update` to update all packages based on the version ranges in `package.json`.
- **Uninstall:**
  - `npm uninstall [package-name]` to remove a dependency from `node_modules`.
  - To also remove it from `package.json`, use `npm uninstall [package-name] --save` (for dependencies) or `npm uninstall [package-name] --save-dev` (for devDependencies).

---

3. Asynchronous Programming

21. Difference between synchronous and asynchronous in Node.js.

- **Synchronous (blocking):** Operations are executed one after the other. The program waits for one operation to complete before moving to the next. This can be slow for I/O-intensive tasks.
- **Asynchronous (non-blocking):** Operations are initiated, and the program continues executing without waiting for them to finish. A **callback function** is provided to handle the result when the operation completes. Node.js's I/O operations are almost always asynchronous.

22. What are callbacks? Problems with callbacks (callback hell).

A **callback** is a function passed as an argument to another function. The receiving function then "calls back" the provided function when an asynchronous operation is complete.

Callback hell (also known as the pyramid of doom) is a situation where nested callbacks become so deep and complex that the code is difficult to read, understand, and maintain.

23. Difference between Promises and Callbacks.

- **Callbacks:** The traditional approach. They are often a function passed as an argument. The nesting can lead to callback hell.
- **Promises:** A modern alternative. A **Promise** is an object representing the eventual completion or failure of an asynchronous operation. It has three states: **pending**, **fulfilled**, or **rejected**. Promises allow for cleaner, chainable asynchronous code using `.then()` and `.catch()` methods.

24. What is async/await in Node.js?

`async/await` is an even more modern syntactic sugar built on top of Promises. It allows you to write asynchronous code that looks and behaves like synchronous code, making it much more readable.

- The `async` keyword is used to declare an asynchronous function.
- The `await` keyword is used to pause the execution of an `async` function until a Promise is resolved.

25. What is the event-driven architecture in Node.js?

In an **event-driven architecture**, components of the system communicate by emitting and listening for **events**. A component that performs an action (the **emitter**) emits an event. Other components (the **listeners**) can register to be notified when that event occurs. Node.js uses this model heavily with its core **EventEmitter** class and the event loop.

26. What is `process.nextTick()`?

`process.nextTick()` is a function that defers the execution of a callback until the **next iteration of the event loop**, right before the I/O event handlers are called. It's part of the **microtask queue**. It's useful for ensuring that a function has a chance to execute after some code but before I/O.

27. Difference between `setImmediate()` and `setTimeout()`.

- **`setImmediate()`**: Executes a callback in the **next phase of the event loop**.
- **`setTimeout(callback, 0)`**: Schedules a callback to be executed after a minimum delay of 0 milliseconds. It gets placed in the **timers phase** of the event loop.

Generally, `setImmediate` will execute before `setTimeout(..., 0)` because the `setImmediate` check happens before the timers check in the event loop's cycle.

28. What are microtasks and macrotasks in Node.js?

The event loop manages different queues for tasks.

- **Microtasks** are tasks with higher priority. They are executed after the current synchronous code block is finished and before the event loop proceeds to the next phase. Examples include **Promise callbacks** (`.then()`, `.catch()`, `.finally()`) and `process.nextTick()`.
- **Macrotasks** are lower-priority tasks that are executed in subsequent phases of the event loop. Examples include `setTimeout()`, `setImmediate()`, and I/O callbacks.

29. How does the event loop handle async functions?

When an `async` function is called, it returns a Promise. When it encounters an `await` keyword, it pauses its execution and places the rest of the function's code in a **microtask queue**. The event loop continues processing other code. Once the awaited Promise is resolved, the paused function is taken from the microtask queue and resumes execution.

30. What is `Promise.all` vs `Promise.race` vs `Promise.allSettled`?

- `Promise.all`: Takes an array of promises and returns a single Promise that **resolves** when **all** of the input promises have resolved, returning an array of their resolved values. It **rejects** as soon as any one of the promises rejects.
- `Promise.race`: Takes an array of promises and returns a single Promise that **resolves or rejects** as soon as **any** of the input promises resolves or rejects.
- `Promise.allSettled`: Takes an array of promises and returns a single Promise that **resolves** when **all** of the input promises have settled (either resolved or rejected). It returns an array of objects describing the outcome of each promise.

---

4. File System & Streams

31. What is the `fs` module in Node.js?

The `fs` (File System) module is a core Node.js module that provides an API for interacting with the file system. It offers a variety of methods for reading, writing, creating, and deleting files and directories.

32. Difference between synchronous and asynchronous file operations.

- **Synchronous (`fs.readFileSync`):** The function blocks the execution of the program until the file operation is complete. This can be problematic in a single-threaded environment as it can freeze the application.
- **Asynchronous (`fs.readFile`):** The function returns immediately, and the program continues. A callback function is executed when the file operation is complete. This is the preferred method in Node.js to prevent blocking the event loop.

33. What are streams in Node.js? Types of streams.

Streams are abstract interfaces for handling data flow, a continuous flow of data from one source to another. They are more efficient than traditional file operations for large files because they process data in **chunks**, reducing memory usage.

There are four types of streams:

- **Readable Streams:** For reading data from a source (e.g., `fs.createReadStream`).
- **Writable Streams:** For writing data to a destination (e.g., `fs.createWriteStream`).
- **Duplex Streams:** Both readable and writable (e.g., `net.Socket`).
- **Transform Streams:** A type of Duplex stream that can modify data as it's being read and written (e.g., `zlib.createGzip`).

34. Difference between `readFile` and `createReadStream`.

- **`fs.readFile`**: Reads the **entire file into memory** at once. This is suitable for small files but can be inefficient and lead to memory issues for large files.
- **`fs.createReadStream`**: Reads the file in **chunks**. It's ideal for large files as it uses less memory and allows you to start processing the data as soon as the first chunk arrives.

35. How to handle backpressure in streams?

Backpressure occurs when a **Writable stream** cannot handle data as fast as a **Readable stream** is providing it. To handle it, you can:

- Use the **`pipe()`** method, which automatically manages the flow and handles backpressure.
- Manually listen to the **`drain`** event on the Writable stream to resume reading from the Readable stream.

36. What is piping in Node.js streams?

Piping is a mechanism for connecting the output of a Readable stream to the input of a Writable stream. It simplifies data transfer and automatically handles backpressure. The syntax is simple: `readableStream.pipe(writableStream)`.

37. How to read and write JSON files in Node.js?

- **Read:** Use `fs.readFile()` to read the file, and then use `JSON.parse()` to convert the JSON string into a JavaScript object.
- **Write:** Use `JSON.stringify()` to convert a JavaScript object to a JSON string, and then use `fs.writeFile()` to write the string to a file.

38. Difference between `Buffer` and `Stream`.

- **`Buffer`**: A **temporary storage area** for raw binary data. It's a fixed-size memory allocation used to handle data like images, audio, or encrypted files.
- **`Stream`**: An abstract interface for **continuous data flow**. It processes data in chunks, making it suitable for large files and real-time data transfer. A Buffer is often used as a container for a single chunk of data within a stream.

39. How to watch file changes in Node.js?

You can use the `fs.watch()` or `fs.watchFile()` methods from the `fs` module to monitor a file or directory for changes.

- **`fs.watch()`**: More efficient as it uses OS-specific file system change notifications.
- **`fs.watchFile()`**: Polls the file for changes, which can be less efficient and more resource-intensive.

40. What is zlib in Node.js? (compression).

The **`zlib`** module is a core Node.js module that provides compression and decompression functionality using **gzip** and **deflate**. It can be used for tasks like compressing and decompressing HTTP requests and responses, or for archiving and unarchiving files.

---

5. Networking & APIs

41. How to create a simple HTTP server in Node.js?

You can create a simple HTTP server using the built-in `http` module.

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

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

```
server.listen(3000, '127.0.0.1', () => {
  console.log('Server is running at http://127.0.0.1:3000/');
});
```

42. Difference between `http` and `https` modules.

- `http`: The core module for creating unencrypted HTTP servers.
- `https`: The core module for creating **secure, encrypted HTTPS servers**. It requires an **SSL certificate and a private key** for secure communication. It's essential for production applications to ensure data privacy and security.

43. How to handle GET and POST requests in Node.js?

You can handle different HTTP methods by checking the `req.method` property inside the server's request listener.

- **GET:** Data is sent in the URL query string.
- **POST:** Data is sent in the request body. You must listen for `data` events and `end` events on the request stream to collect the body data.

44. What is Express.js? Why use it?

Express.js is a **minimalist, flexible Node.js web application framework**. It simplifies the process of building web applications and APIs by providing a robust set of features, including routing, middleware, and request/response handling. It's a de-facto standard for building web servers with Node.js.

45. Difference between Express.js and Node.js.

- **Node.js** is the **runtime environment**. It provides the core functionality to run JavaScript on the server.
- **Express.js** is a **framework** built on top of Node.js. It adds a layer of abstraction and provides helpful utilities to make building web applications much faster and more organized.

46. What are middlewares in Express? Types of middleware.

Middleware are functions that have access to the request (`req`), response (`res`), and the next middleware function (`next`) in the application's request-response cycle. They can modify the request and response objects, execute code, and end the cycle.

Types of middleware:

- **Application-level middleware:** Used with `app.use()` and `app.get()`.
- **Router-level middleware:** Used with `express.Router()` for modular routing.

- **Built-in middleware:** Comes with Express, like `express.static()` for serving static files.
- **Error-handling middleware:** Has four arguments (`err, req, res, next`) for handling errors.
- **Third-party middleware:** Installed via NPM (e.g., `body-parser`, `cors`).

47. How to handle routing in Express?

Routing in Express refers to defining how an application responds to a client request to a specific endpoint (a URI and a specific HTTP method). You use methods like `app.get()`, `app.post()`, `app.put()`, and `app.delete()` to define routes. Express also has `express.Router()` for creating modular, mountable route handlers.

48. How to parse request body in Express? (`body-parser`, `express.json()`)

The request body doesn't come parsed by default. To parse it:

- **`body-parser`**: A popular third-party middleware used to parse the request body.
- **`express.json()`**: A built-in middleware since Express 4.16.0 that parses incoming JSON requests. It's the modern, preferred way to handle JSON bodies.

49. Difference between `res.send()` and `res.json()`.

- **`res.send()`**: A versatile method that sends a response of various types. It automatically sets the `Content-Type` header based on the data type.
- **`res.json()`**: A specialized method that sends a **JSON response**. It explicitly sets the `Content-Type` header to `application/json` and converts the provided object to a JSON string. It's more specific and often preferred for API development.

50. How to handle file uploads in Node.js? (Multer).

Handling file uploads can be complex. The most common solution is to use a third-party middleware like **Multer**. Multer is a middleware for handling `multipart/form-data`, which is primarily used for uploading files. It simplifies the process by parsing the incoming request and making the uploaded files available on the `req.files` or `req.file` object.

---

6. Databases with Node.js

51. How to connect Node.js with MongoDB?

You can connect to MongoDB from Node.js using a **driver** or an **ODM** (Object Document Mapper).

- **Driver:** The official `mongodb` package provides low-level access.
- **ODM: Mongoose** is the most popular ODM. It provides a higher-level, schema-based solution for modeling your application data.

52. Difference between MongoDB and SQL databases in Node.js context.

- **SQL (e.g., MySQL, PostgreSQL):**
  - **Relational:** Data is stored in tables with predefined schemas.
  - **Structured:** Requires a fixed schema.
  - **Transaction-based:** Supports complex transactions with ACID properties.
  - **Use Case:** Applications where data integrity and structured relationships are critical.
- **MongoDB (NoSQL):**
  - **Document-based:** Data is stored in JSON-like documents.
  - **Flexible:** Schemaless, allowing for flexible data models.
  - **Use Case:** High-volume, real-time applications where data structures can evolve rapidly.

53. How to connect Node.js with MySQL/PostgreSQL?

You can connect to SQL databases using a **database driver** or an **ORM** (Object-Relational Mapper).

- **Driver:** Packages like `mysql` or `pg` for direct queries.
- **ORM: Sequelize** and **TypeORM** are popular choices. They allow you to interact with the database using object-oriented syntax rather than raw SQL queries.

54. What is Mongoose in Node.js?

Mongoose is an **Object Data Modeling (ODM) library** for MongoDB and Node.js. It provides a schema-based solution for data modeling, validation, and querying. It allows you to define a **schema** for your data, which gives your documents structure and enforces validation.

55. Difference between Mongoose and native MongoDB driver.

- **Native MongoDB Driver:** Provides low-level, direct access to MongoDB's API. It's a thin layer over the database, offering maximum flexibility and control but requiring more manual work for validation and data modeling.
- **Mongoose:** A higher-level abstraction. It sits on top of the native driver and adds features like schema validation, middleware, and a more user-friendly API for common operations. It's often preferred for building applications quickly.

56. How to handle database connection pooling in Node.js?

Connection pooling is a technique that keeps a set of open database connections ready for use. Instead of creating a new connection for every request, the application reuses an

existing one from the pool. This reduces latency and resource overhead. Most modern database drivers and ORMs handle connection pooling automatically.

57. What are ORMs in Node.js (Sequelize, TypeORM)?

An **ORM** (Object-Relational Mapper) is a library that allows you to interact with a relational database using an object-oriented paradigm. Instead of writing raw SQL queries, you interact with database tables as JavaScript objects.

- **Sequelize:** A popular promise-based ORM that supports MySQL, PostgreSQL, SQLite, and more.
- **TypeORM:** A powerful ORM that supports `Active Record` and `Data Mapper` patterns and works well with TypeScript.

58. How to handle transactions in Node.js databases?

- **SQL:** Transactions are a core concept in SQL. Most ORMs (like Sequelize) provide an easy-to-use API for managing transactions with `commit` and `rollback` functionality.
- **MongoDB:** MongoDB supports **multi-document transactions** for replica sets and sharded clusters, which allows you to perform ACID-compliant operations across multiple documents.

59. What is ACID in databases and how does Node.js handle it?

ACID is an acronym for a set of properties that guarantee reliable database transactions:

- **Atomicity:** A transaction is all-or-nothing.
- **Consistency:** A transaction brings the database from one valid state to another.
- **Isolation:** Concurrent transactions don't interfere with each other.
- **Durability:** Once a transaction is committed, it remains committed even if the system fails.

Node.js, as a runtime, doesn't directly handle ACID. Instead, it relies on the underlying database to enforce these properties. SQL databases have native support, while MongoDB has introduced multi-document transactions to provide ACID guarantees.

60. What is indexing in MongoDB? How does Node.js use it?

Indexing in MongoDB is a data structure that helps locate and retrieve data efficiently. Without an index, MongoDB has to scan every document in a collection.

In Node.js with Mongoose, you can define indexes in your schema:

JavaScript
```
const userSchema = new mongoose.Schema({
  name: String,
  email: { type: String, unique: true, index: true },
```

});

This tells MongoDB to create an index on the `email` field, which significantly speeds up queries.

---

7. Security in Node.js

61. How to secure REST APIs in Node.js?

Securing REST APIs involves several layers:

- **Authentication:** Verifying the user's identity (e.g., JWT, OAuth).
- **Authorization:** Granting permissions based on the user's role.
- **Input Validation:** Sanitizing and validating all user input to prevent attacks like SQL Injection and XSS.
- **HTTPS:** Using the `https` module to encrypt all traffic.
- **Rate Limiting:** Protecting against brute-force attacks and abuse.
- **Helmet.js:** A middleware that adds various security headers.

62. What is Helmet.js and why use it?

Helmet.js is a Node.js middleware that helps secure Express applications by setting various HTTP headers. It's a collection of smaller middleware functions that collectively protect against common web vulnerabilities, such as Cross-Site Scripting (XSS), by setting headers like `X-Content-Type-Options` and `X-Frame-Options`.

63. How to prevent SQL Injection in Node.js apps?

SQL Injection is an attack where an attacker inserts malicious SQL code into an input field. To prevent it, you should **never concatenate user input directly into SQL queries**. Instead, use:

- **Prepared statements:** A pre-compiled SQL query with placeholders for parameters.
- **Parameterized queries:** The database driver handles the sanitization of the input.
- **ORMs:** ORMs like Sequelize automatically protect against SQL injection by using prepared statements.

64. How to prevent XSS in Node.js?

Cross-Site Scripting (XSS) is an attack where an attacker injects malicious scripts into a web page viewed by other users. To prevent it:

- **Escape user input:** Sanitize all user-generated content before rendering it.
- **Use HTML sanitizers:** Libraries like `dompurify` can help.

- **Content Security Policy (CSP):** A security header that restricts the sources of content a browser can load. Helmet.js can help with this.

65. What is CSRF? How to prevent it in Node.js?

CSRF (Cross-Site Request Forgery) is an attack where a malicious website tricks a user's browser into performing an unwanted action on a trusted site where the user is authenticated. To prevent it:

- **CSRF Tokens:** A unique, secret token is generated for each user session and included in forms and AJAX requests. The server verifies this token.
- **Double Submit Cookie:** A CSRF token is stored in both a cookie and a hidden form field.

66. Difference between authentication and authorization in Node.js.

- **Authentication:** The process of **verifying the identity** of a user. It answers the question, "Who is this user?" (e.g., verifying a password or a JWT).
- **Authorization:** The process of **determining what a user is allowed to do**. It answers the question, "Does this user have permission to access this resource?" (e.g., checking if a user has an 'admin' role).

67. How JWT (JSON Web Token) works in Node.js?

A **JWT** is a compact, URL-safe means of representing claims to be transferred between two parties. It consists of three parts:

- **Header:** Contains the token type and the signing algorithm.
- **Payload:** Contains the claims (e.g., user ID, roles).
- **Signature:** Created by combining the header, payload, and a secret key.

On the server, you **sign** a JWT and send it to the client. The client stores it (e.g., in `localStorage`). On subsequent requests, the client sends the JWT in the `Authorization` header. The server then **verifies** the token's signature using the secret key to ensure its authenticity.

68. Difference between session-based and token-based authentication.

- **Session-based:**
  - A session ID is stored in a cookie on the client.
  - The server stores the session data (e.g., user info) in its memory or a database.
  - **Stateful:** The server must maintain the session state.
- **Token-based (e.g., JWT):**
  - The client stores the token (e.g., in `localStorage`).
  - The token contains all the necessary user data.
  - **Stateless:** The server doesn't need to store session information. This is ideal for microservices and scalable APIs.

69. How to encrypt passwords in Node.js? (bcrypt).

Never store plain-text passwords. Passwords should be **hashed** using a secure, one-way algorithm like **bcrypt**.

- `bcrypt` is a library that creates a salted hash of the password.
- **Salting** adds random data to the password before hashing, which prevents attacks using pre-computed rainbow tables.

70. How to enable HTTPS in Node.js?

You can enable HTTPS using the built-in `https` module. It requires a **private key** and a **public certificate**.

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

const options = {
  key: fs.readFileSync('path/to/your/private.key'),
  cert: fs.readFileSync('path/to/your/certificate.crt'),
};

https.createServer(options, (req, res) => {
  res.end('Hello, Secure World!');
}).listen(443);
```

---

8. Advanced Node.js

71. What is clustering in Node.js?

Clustering is a technique for running multiple instances of a Node.js application on a single server to take advantage of multi-core systems. The built-in `cluster` module allows you to create a **master process** that forks **worker processes**, with each worker process running on a separate CPU core. This helps with CPU-intensive tasks and can improve performance.

72. What is PM2? Why use it?

PM2 (Process Manager 2) is a popular, production-ready process manager for Node.js applications. It helps you manage your applications in production. Key features include:

- **Automatic clustering:** It can automatically scale your app across all CPU cores.
- **Load balancing:** It distributes incoming traffic among worker processes.
- **Hot reloading:** It can reload your application without downtime.
- **Logging and monitoring:** It provides detailed logs and health metrics.

73. Difference between `spawn`, `fork`, and `exec` in child processes.

All three are methods in the `child_process` module to run external programs.

- **`spawn`**: Spawns a new process asynchronously. It is the most robust way to run a command. It returns a `ChildProcess` object with streams for `stdin`, `stdout`, and `stderr`, allowing you to handle large amounts of data.
- **`exec`**: Spawns a new process and buffers the entire output. It's best for small, simple commands. It can lead to memory issues with large output.
- **`fork`**: A special case of `spawn` designed specifically for forking Node.js processes. It establishes a communication channel between the parent and child processes using the `send()` and `on('message')` methods. This is what the `cluster` module uses.

74. What is `worker_threads` module?

The **`worker_threads`** module, introduced in Node.js 10.5.0, allows for the use of **true multi-threading** in Node.js for CPU-intensive tasks. Unlike the event loop, which handles I/O, worker threads can perform blocking, CPU-intensive work without blocking the main event loop. They use a message-passing system to communicate with the main thread.

75. Difference between process and thread in Node.js.

- **Process:** An independent instance of a running program with its own memory space and resources. In Node.js, each `cluster` worker is a separate process.
- **Thread:** A single sequence of instructions within a process. Multiple threads within a single process share the same memory space. The Node.js event loop runs on a single thread, while `worker_threads` allow you to create additional threads.

76. How does Node.js handle multiple requests with single thread?

Node.js handles multiple requests with a single thread by using a **non-blocking, event-driven architecture**. When a request involves a slow operation (like a database query), Node.js doesn't wait for it to finish. It offloads the task to the underlying system (e.g., using a thread pool for I/O) and continues processing other requests. When the slow operation is complete, its callback is added to the event queue, and the event loop eventually processes it.

77. What is load balancing in Node.js?

Load balancing is the process of distributing incoming network traffic across multiple backend servers to ensure no single server is overwhelmed. In Node.js, you can implement load balancing in a few ways:

- **PM2:** Can automatically handle load balancing with its clustering feature.
- **Reverse Proxy:** Using a server like **Nginx** or **HAProxy** to distribute requests to multiple Node.js instances.

78. How to handle concurrency in Node.js?

Concurrency is the ability to handle multiple tasks at the same time. Node.js handles it primarily through its **event loop** and **non-blocking I/O model**. For CPU-bound tasks, concurrency can be handled by:

- **Clustering:** Spawning multiple processes to utilize multi-core systems.
- **Worker Threads:** Using the `worker_threads` module to run CPU-intensive tasks in a separate thread.

79. What is WebSocket in Node.js?

WebSockets provide a persistent, full-duplex communication channel between a client and a server over a single TCP connection. Unlike HTTP, which is stateless, WebSockets allow for real-time, bi-directional communication, making them ideal for applications like chat apps, real-time dashboards, and multiplayer games. Libraries like `socket.io` simplify using WebSockets in Node.js.

80. Difference between REST and WebSockets in Node.js.

- **REST:**
  - **Protocol:** HTTP.
  - **Communication:** Request-response model. The client sends a request, and the server sends a response.
  - **Stateless:** Each request is independent.
  - **Use Case:** CRUD operations for APIs.
- **WebSockets:**
  - **Protocol:** TCP.
  - **Communication:** Full-duplex, bi-directional. Both the client and server can send messages at any time.
  - **Stateful:** A persistent connection is maintained.
  - **Use Case:** Real-time applications requiring instant updates.

---

9. Testing & Deployment

81. How to test Node.js applications? (Mocha, Jest).

Testing is a critical part of development. Common testing frameworks include:

- **Jest:** A popular, all-in-one testing framework from Facebook. It includes a test runner, assertion library, and mocking capabilities. It's often the default choice for modern applications.
- **Mocha:** A flexible testing framework. It requires a separate assertion library (like `Chai`) and a mocking library (like `Sinon`).

82. Difference between unit testing and integration testing in Node.js.

- **Unit Testing:** Tests individual components (functions, modules) in isolation. The goal is to verify that each component works correctly. You typically **mock** external dependencies (e.g., database calls) to ensure the test is isolated.
- **Integration Testing:** Tests how different components or modules work together as a group. The goal is to verify that the integrated system behaves as expected. You typically test the entire flow (e.g., from an API endpoint to a database).

83. How to mock database calls in Node.js tests?

Mocking is the process of replacing real objects with fake, controlled versions. To mock database calls in tests, you can use a mocking library like `Sinon` or `Jest's` built-in mocking capabilities. This allows you to test your business logic without actually connecting to a database, making your tests faster and more reliable.

84. What is Supertest in Node.js?

Supertest is a library for testing HTTP requests in Node.js. It's often used for **integration testing** of Express or other HTTP servers. It allows you to simulate HTTP requests and make assertions about the response, such as status codes, headers, and body content.

85. What is CI/CD in Node.js deployment?

CI/CD stands for **Continuous Integration/Continuous Deployment**. It's a practice of automating the software delivery process.

- **CI (Continuous Integration):** Developers merge their code changes into a central repository, and an automated build and test process runs.
- **CD (Continuous Deployment):** If the tests pass, the changes are automatically deployed to a production environment.

86. Difference between Dockerizing and deploying Node.js on servers.

- **Deploying on a Server:** Involves manually installing Node.js, dependencies, and managing the application on a server (e.g., using PM2). This can lead to environmental inconsistencies between development and production.
- **Dockerizing:** Involves packaging the application and its dependencies into a self-contained unit called a **Docker container**. This ensures the application runs consistently across different environments, from development to production.

87. What is environment variable handling in Node.js (`dotenv`)?

Environment variables are key-value pairs that are external to your application code. They are used to store configuration data that can change between environments (e.g., database connection strings, API keys). The **`dotenv`** library helps load these variables from a `.env` file into `process.env`. This keeps sensitive information out of your codebase.

88. How to monitor Node.js applications? (New Relic, PM2 logs).

Monitoring is crucial for understanding an application's health and performance.

- **PM2:** Provides built-in monitoring with commands like `pm2 logs` and `pm2 monit`.
- **APM (Application Performance Monitoring) tools:** Services like **New Relic**, **Datadog**, and **Sentry** provide in-depth insights into application performance, error rates, and resource usage.

89. How to debug Node.js applications?

- **`console.log()`:** The simplest but often least effective way.
- **`node --inspect`:** The built-in debugger. You can attach a debugger (e.g., from VS Code or Chrome DevTools) to your running Node.js process to set breakpoints, inspect variables, and step through code.
- **IDE Debuggers:** Modern IDEs like VS Code have excellent built-in debugging tools.

90. Difference between production and development builds in Node.js.

- **Development Build:** Optimized for speed and debugging. It often includes source maps and is not minified. Environment variables (`NODE_ENV`) are set to `development`.
- **Production Build:** Optimized for performance and security. Code is minified, unnecessary modules are removed, and sensitive environment variables are loaded. Error messages are often generic to prevent information leaks.

---

10. Real-world & System Design

91. How to implement rate limiting in Node.js?

Rate limiting is a technique to control the number of requests a user can make to an API over a period of time. It protects against brute-force attacks and abuse. You can implement it using:

- **Middleware:** Use libraries like `express-rate-limit`.
- **In-memory cache:** Use a simple JavaScript object or a library like `lru-cache` to store request counts.
- **Dedicated services:** Use services like **Redis** for distributed rate limiting.

92. What is caching in Node.js? (Redis).

Caching is the process of storing frequently accessed data in a temporary, fast-access memory store to reduce the number of times you have to fetch it from a slower source (e.g.,

a database). **Redis** is a popular in-memory data store often used for caching due to its high performance and versatility.

93. Difference between Redis and Memcached for Node.js caching.

- **Redis:** More than just a cache. It's a feature-rich data structure store. It supports various data types (strings, lists, sets, hashes) and offers features like persistence and pub/sub messaging.
- **Memcached:** A simpler, high-performance, in-memory key-value store. It's optimized purely for caching. It's often faster for simple key-value lookups but lacks the advanced features of Redis.

94. How to implement queues in Node.js? (Bull, RabbitMQ, Kafka).

Queues are used to handle asynchronous, time-consuming tasks outside the main request-response cycle. They improve performance by offloading heavy work.

- **Bull:** A simple, high-performance library for using **Redis** as a job queue.
- **RabbitMQ:** A robust message broker that provides advanced queuing features.
- **Kafka:** A distributed streaming platform that's ideal for building real-time data pipelines.

95. What is EventEmitter in Node.js?

The **EventEmitter** is a class in Node.js that provides an API for creating and handling custom events. It is a cornerstone of Node.js's event-driven architecture. Objects that emit events are instances of `EventEmitter`. You can use `emitter.on()` to listen for an event and `emitter.emit()` to fire it.

96. How to handle large file uploads in Node.js?

Handling large file uploads efficiently requires using **streams**. Instead of buffering the entire file into memory, you can read the file in chunks and stream it directly to a destination, such as a cloud storage service like Amazon S3. Libraries like **Multer** and **busboy** can handle this.

97. How to scale Node.js applications in microservices?

Microservices is an architectural approach where an application is composed of small, independent, and loosely coupled services. Node.js is a great fit for this due to its lightweight nature.

- **API Gateway:** A single entry point for all client requests.
- **Service Discovery:** Services can find and communicate with each other.
- **Message Queues:** Used for asynchronous communication between services.

98. What are design patterns in Node.js? (Singleton, Factory, Observer).

Design patterns are reusable solutions to common software design problems.

- **Singleton:** Ensures a class has only one instance and provides a global point of access to it.
- **Factory:** Provides an interface for creating objects without specifying their exact class.
- **Observer:** Defines a one-to-many dependency between objects, so that when one object (the subject) changes state, all of its dependents (observers) are notified and updated automatically. `EventEmitter` is a good example of this.

99. Difference between monolithic and microservices architecture in Node.js.

- **Monolithic:**
  - A single, large, and tightly coupled application.
  - **Pros:** Simpler to develop and deploy initially.
  - **Cons:** Difficult to scale, maintain, and can be a single point of failure.
- **Microservices:**
  - An application composed of small, independent services.
  - **Pros:** Highly scalable, flexible, and resilient.
  - **Cons:** More complex to develop, deploy, and manage.

100. What are common performance bottlenecks in Node.js and how to fix them?

- **CPU-intensive operations:** These block the single thread. Fix by using **clustering** or **worker threads** to offload the work.
- **Synchronous I/O:** Avoid synchronous file operations (`fs.readFileSync`) as they block the event loop. Use **asynchronous** counterparts.
- **Memory leaks:** Caused by unreleased memory. Use monitoring tools to identify them.
- **Blocking code:** Any long-running, synchronous function will block the event loop. **Profile your code** to find slow functions and refactor them to be non-blocking.