1. **Why Express js is used ?**

Express.js is a popular choice for web development due to its:

Simplicity: Minimalistic and easy to learn, allowing quick setup and development.

Flexibility: No strict structure, letting you design your application as needed.

Middleware: Easy addition of functionalities through middleware functions.

Robust Routing: Powerful and flexible routing system.

Integration: Seamlessly integrates with other Node.js libraries and frameworks.

Performance: Built on Node.js, ensuring high performance and scalability.

Community Support: Large community with extensive plugins, tools, and resources.

Express.js enables efficient and scalable web application development with a supportive ecosystem and straightforward approach.

1. **What is .env file used for?**

The .env file is used for storing sensitive information in a web application which we don’t want to expose to others like password, database connection string etc. It is a simple text file where each line represents a key-value pair, and these pairs are used to configure various aspects of the application.

1. **What is Authorization in Nodejs and Authentication in Node JS?**

It determines whether the authenticated user has permission to access a resource or perform an action. It answers the question: *What are you allowed to do?*

**Understanding the Authorization:**

Authorization is the process of determining what an authenticated user or system is allowed to do. It defines permissions and access levels within the system.

**How It Works**:

* **Permissions**: Once authenticated, the system checks the user's permissions or roles to determine what resources or actions they are allowed to access or perform.
* **Access Control**: Based on these permissions, the system grants or restricts access to specific resources or actions.

Authentication is the process of verifying the identity of a user or system. It confirms that a user is who they claim to be.

In web applications, this is typically done through either

 It verifies the identity of a user or system. Essentially, it answers the question: *Who are you?*

stateful (session-based) or

stateless (token-based) methods.

Stateful Authentication:

* In stateful authentication, the server creates a session for the user after successful login and stores session information on the server-side. The server then sends a cookie containing the session ID to the user’s browser, which is stored and sent back with each subsequent request to the server.
* Stateful session is created on the backend side, and the corespondent session reference Id is sent to the client. Each time the client makes a request to the server, the server locates the session memory using the reference Id from the client and finds the authentication information.
* In this model, you can easily imagine that if the session memory is deleted on the backend side, then the session reference Id, which the client is holding, is completely meaningless.

Here’s a basic example of how you might set up session-based authentication in Node.js using the express-session middleware:

const express = require('express');

const session = require('express-session');

const app = express();

app.use(session({

  secret: 'your-secret-key',

  resave: false,

  saveUninitialized: true,

  cookie: { secure: true }

}));

app.post('/login', (req, res) => {

  // Authentication logic...

  if (authenticated) {

    req.session.userId = user.id;

  }

});

Stateless Authentication

In stateless authentication, the server doesn’t store any session data. Instead, it uses tokens (like JWT) which are sent to the client after successful login. The client then sends this token in the Authorization header for each subsequent request.

## **4) What is HTTP Headers?**

The **HTTP headers** are used to pass additional information between the clients and the server through the **request** and **response**header.0020In API calls, headers are essentially additional information attached to a request or response. You can think of them as little envelopes containing details that help the server understand your request better and respond accordingly.

Types of Api Header ?

Two Types of API Header

**Request Header:**

These are sent by the client to the server along with the request. They provide details about what you re asking for and how you want it delivered. Sent from frontend to backend.

**Response Header:**

These are sent back by the server in the response. They contain information about the response itself, such as its format or any error messages.

## **What are the most common HTTP headers?**

HTTP headers play a crucial role in server and client behavior throughout the request and response cycle. Request headers are sent by the client to the server and contain information and instructions related to the requested resource, while response headers are sent by the server to the client and provide metadata, instructions, and additional information about the response itself.

**Some of the most commonly used request headers are**:

### Accept

The Accept header defines the media types that the client is able to accept from the server. For instance, Accept: application/json, text/html indicates that the client prefers [JSON](https://blog.postman.com/what-is-json/) or HTML responses. This information allows the server to send a resource representation that meets the client’s needs.

### User-Agent

The User-Agent header identifies the web browser or client application that is making the request, which enables the server to tailor its response to the client. For instance, if the User-Agent header indicates that the request is coming from the Chrome browser, the server may include CSS prefixes for CSS properties that are compatible with Chrome.

### Authorization

The Authorization header is used to send the client’s credentials to the server when the client is attempting to access a protected resource. For instance, the client might include a [JSON Web Token](https://blog.postman.com/what-is-jwt/) (JWT) as the value of the header, which the server will then verify before returning the requested resource.

### Content-Type

The Content-Type header identifies the media type of the content in the request body. For instance, Content-Type: application/json indicates that the request body contains JSON data. This information helps the server successfully interpret and process the payload.

### Cookie

The client can use the Cookie header to send previously stored cookies back to the server. The server then uses these cookies to associate the request with a specific user or session. This header plays an important role in delivering personalized experiences, as it enables the server to remember a user’s login state or language preference.

**The most common response headers are:**

### Content-Type

The Content-Type response header is the counterpart of the Content-Type request header, as it indicates the type of data that the server is sending to the client. The header value typically includes the media type (such as text/html, application/json, image/jpeg, and audio/mp3), as well as any optional parameters.

### Cache-Control

The Cache-Control header controls caching behavior in the client’s browser or intermediate caches. It defines how the response can be cached, when it expires, and how it should be revalidated. For example, Cache-Control: max-age=3600, public instructs the client to cache the response for a maximum of 3600 seconds (1 hour) and allows caching by public caches.

### Server

The Server header includes the name and version of the server software that generated the response, as well as information about the server’s technology stack. For instance, Server: Apache/2.4.10 (Unix) indicates that the response was generated by the Apache web server version 2.4.10. It’s important to note that the Server header is informational and doesn’t affect the API’s functionality.

### Set-Cookie

The Set-Cookie header instructs the client to store a cookie with the specified name, value, and additional attributes, such as expiration, domain, path, and security flags. The client will then include the cookie in subsequent requests in order to facilitate stateful communication and personalized experiences.

### Content-Length

The Content-Length header, which specifies the size of the response body in bytes, can help the client anticipate how much data it is going to receive. This improves performance by allowing the client to plan in advance for more efficient memory allocation and data processing.

[**Which are the arguments available to an Express JS route handler function?**](https://www.geeksforgeeks.org/express-js-router-param-function/)

In Express JS route handler function, there are mainly3 arguments available that provide useful information and functionality.

* req: This represents the HTTP request object which holds information about the incoming request. It allows you to access and manipulate the request data.
* res: This represents the HTTP response object which is used to send the response back to the client. It provides methods and properties to set response headers, status codes, and send the response body.
* next: This is a callback function that is used to pass control to the next middleware function in the request-response cycle.

[**How can you deal with error handling in Express.js?**](https://www.geeksforgeeks.org/explain-error-handling-in-express-js-using-an-example/)

Express.js provides built-in error-handling mechanism with the help of the next() function. When an error occurs, you can pass it to the next middleware or route handler using the next() function. You can also add an error-handling middleware to your application that will be executed whenever an error occurs.

1. **What is Server side Rendering ?**

Server-Side Rendering (SSR) is a technique used in web development where the server generates the HTML of a web page and sends it to the client's browser. This ensures that the client receives a fully rendered page, which can be displayed immediately.

Server-side rendering is exactly what it sounds like: rendering on the server. When you do this, you'll create an HTML file with all of the content of the site and send it back to the user. The user will then get an fully rendered HTML page that contains all of the necessary information for them to see your site without having to wait for any JavaScript or CSS files to load. This means that users who visit your site will be able to see everything much faster than if they were just looking at blank screen while waiting for JavaScript files to load.

**Benefits of SSR**

Improved Performance:

Initial Load Time: Since the server sends a fully rendered HTML page, users can see content immediately without waiting for JavaScript to render the page.

Perceived Performance: Faster initial load times improve the user experience, making the site feel more responsive.

How SSR Works

**Client Request:**

The user navigates to a URL in their browser, which sends a request to the server.

.

**Hydration:**

Hydration is the process where the JavaScript framework (like React, Vue, or Angular) takes over the static HTML, attaches event listeners, and makes the page interactive.

During hydration, the client-side JavaScript reconciles the server-rendered HTML with its virtual DOM to add interactivity without re-rendering the entire page.

};

## What is JSON Web Token?

JSON Web Token (JWT) defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the **HMAC** algorithm) or a public/private key pair using **RSA** or **ECDSA**. This also allows the Stateless authentication.

Although JWTs can be encrypted to also provide secrecy between parties, we will focus on signed tokens. Signed tokens can verify the integrity of the claims contained within it, while encrypted tokens hide those claims from other parties. When tokens are signed using public/private key pairs, the signature also certifies that only the party holding the private key is the one that signed it.

In .ENV secret key and expiration data claims

**What is the JSON Web Token structure?**

In its compact form, JSON Web Tokens consist of three parts separated by dots (.), which are:

* Header
* Payload
* Signature

Therefore, a JWT typically looks like the following.

xxxxx.yyyyy.zzzzz

Let's break down the different parts.

**Header**

The header *typically* consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.

For example:

{

"alg": "HS256",

"typ": "JWT"

}

Then, this JSON is **Base64Url** encoded to form the first part of the JWT.

**Payload**

The second part of the token is the payload, which contains the claims. Claims are statements about an entity (typically, the user) and additional data. There are three types of claims: *registered*, *public*, and *private* claims.An example payload could be:

{

"sub": "1234567890",

"name": "John Doe",

"admin": true

}

The payload is then **Base64Url** encoded to form the second part of the JSON Web Token.

Do note that for signed tokens this information, though protected against tampering, is readable by anyone. Do not put secret information in the payload or header elements of a JWT unless it is encrypted.

**Signature**

To create the signature part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.

For example if you want to use the HMAC SHA256 algorithm, the signature will be created in the following way:

HMACSHA256(

base64UrlEncode(header) + "." +

base64UrlEncode(payload),

secret)

**The signature is used to verify the message wasn't changed along the way, and, in the case of tokens signed with a private key, it can also verify that the sender of the JWT is who it says it is.**

**How JWT Works**

**1. Authentication Flow**

1. User logs in with credentials.
2. Server validates credentials and generates a JWT.
3. JWT is sent to the client (usually stored in localStorage, sessionStorage, or a cookie).
4. Client includes JWT in the Authorization header of future requests:

makefile

Copy code

Authorization: Bearer <JWT>

**2. Verification Flow**

1. Server receives the JWT from the client.
2. Server verifies the token's signature and decodes it.
3. If valid, the server processes the request based on the claims.

**What is the difference between accessToken and refreshToken in JWT?**

* **Answer**:
  + accessToken: Short-lived, used to access protected resources.
  + refreshToken: Long-lived, used to obtain a new accessToken without re-authentication.

 **How can you prevent JWT from being stolen?**

* **Answer**:
  + Use HTTPS.
  + Store JWT in httpOnly cookies.
  + Implement token expiration and refresh tokens.

 **What is the role of the signature in a JWT?**

* **Answer**: Ensures the integrity and authenticity of the token.

 **How do you handle token expiration in JWT?**

* **Answer**: Include the exp claim and generate a new token using a refresh token.

 **What are the security concerns with JWT?**

* **Answer**:
  + Weak secret keys.
  + Storing tokens in localStorage (vulnerable to XSS attacks).
  + Missing token expiration.

 **Explain the difference between symmetric and asymmetric signing in JWT.**

* **Answer**:
  + **Symmetric**: Uses a single secret key (e.g., HMAC).
  + **Asymmetric**: Uses a key pair (public/private) (e.g., RSA/ECDSA).

 **How would you apply JWT for authentication and authorization?**

* **Authentication**: Upon user login, the server verifies the credentials and issues a JWT containing user claims (e.g., user\_id, roles). The token is sent back to the client and stored securely (e.g., in an httpOnly cookie).
* **Authorization**: The client includes the JWT in the Authorization header of subsequent requests (Bearer <token>). The server validates the token and extracts claims to determine the user's permissions for accessing resources.

**What techniques do you use to ensure secure usage of JWT?**

* Use httpOnly and secure cookies to store JWTs and prevent XSS attacks.
* Always use HTTPS to prevent MITM (Man-in-the-Middle) attacks.
* Implement short-lived accessTokens and use refreshTokens for re-authentication.
* Use strong secret keys for symmetric signing or private keys for asymmetric signing.
* Validate the token’s expiration (exp) and issued-at (iat) claims.
* Regularly rotate keys and implement proper key management.

In a JSON Web Token (JWT), **claims** are pieces of information embedded within the token payload that represent statements about an entity (usually the user) and additional metadata. Claims are the core content of a JWT and serve to convey information between parties securely.

**7) Process of Handling Requests**

* Receives a request → Node.js accepts the request on its single thread.
* Checks if it's blocking or non-blocking:
* Non-blocking tasks (e.g., database queries, file reading) are offloaded to the Thread Pool in the background.
* Blocking tasks (e.g., heavy calculations) can use Worker Threads to avoid slowing down the main thread.
* Processes other requests while waiting → Node.js doesn’t wait for the first task to complete; it keeps handling other incoming requests.
* Completes the task and sends the response → Once a background task is done, the callback function is triggered to return the response.

https://rabisiddique.medium.com/maximizing-node-js-performance-with-thread-pools-912bacbe529a

2) setImmediate() Executes after I/O events (in the next event loop cycle).

setTimeout() Executes after a specific delay.

process.nextTick() Executes immediately after the current operation, before I/O events

3) What are Streams in Node.js?

Answer:

Streams are used to handle large amounts of data efficiently. They process data in chunks instead of loading everything into memory.

Types of Streams:

Readable Streams → Reading data (e.g., fs.createReadStream())

Writable Streams → Writing data (e.g., fs.createWriteStream())

Duplex Streams → Both read & write (e.g., TCP sockets)

Transform Streams → Modify data while reading/writing (e.g., compression)

4) Does Node.js offload non-blocking tasks to the thread pool?

"Is the task CPU-bound or I/O-bound? If it's CPU-bound (like crypto), it uses the thread pool. If it's I/O-bound (like HTTP requests), the OS handles it."

5) What happens if a heavy computation task runs in the main thread?

Use Worker Threads: Offload heavy tasks to separate threads.

Use Child Processes: Run CPU-heavy operations in a separate process.

Use Queues (like Redis): Process jobs asynchronously.

6) What is the difference between Clustering and Load Balancing?

Answer:

✅ Clustering (cluster module in Node.js):

Cluster allows you to run multiple instances of Node.js, utilizing all CPU cores for better performance.

Spawns multiple Node.js processes, each running on a different core

Each worker runs independently and handles requests, but state is not shared between workers."

Trick: Workers don’t share memory. Use Redis or a shared database for communication.

Load Balancing (via Nginx or PM2):

Distributes requests across multiple servers.

Uses round-robin, least connections, IP hashing, etc.

Key Difference:

Clustering runs multiple Node.js instances on one machine.

Load Balancing distributes traffic across multiple machines.

7) What is the difference between streaming and buffering in Node.js?

Answer:

✔ Buffering: Reads entire file into memory before processing.

✔ Streaming: Reads data in chunks, using less memory.

8) How do you prevent race conditions in async code?

Race conditions in asynchronous code occur when multiple operations access shared resources in an unpredictable order, leading to inconsistent or incorrect behavior. Here are some ways to prevent them:

1. Using Locks (Mutexes)

Implement a locking mechanism to ensure that only one operation modifies the shared resource at a time.

9) **What is the Order of Middleware Execution in Express?**

In **Express.js**, middleware functions execute in the **order they are defined** in the code. Express uses a **top-to-bottom**, **sequential flow**.

**Order of Execution:**

1. **Global Middleware** (applies to all routes)
2. **Route-Specific Middleware**
3. **Route Handler**
4. **Error Handling Middleware** (only runs if next(err) is called)

10) **How Do You Manage Sessions and Cookies in Express?**

To manage **sessions** and **cookies** in Express, you use:

* cookie-parser for reading cookies
* express-session for session management

**How It Works:**

* **Session** data is stored server-side.
* A **session ID** is stored in a **cookie** on the client.
* On each request, the server uses the session ID from the cookie to retrieve session data.

How do you implement file uploads (e.g., with multer) in Express?

In Express, I use the multer middleware to handle file uploads. Multer is specifically designed to handle multipart/form-data, which is the encoding used when files are uploaded via forms or APIs.

First, I install it with npm install multer, then configure a storage engine. Multer supports two types: diskStorage to store files on the server and memoryStorage to store them in memory.

I usually go with diskStorage for saving files locally. I set the destination and generate unique filenames using timestamps or UUIDs to prevent overwrites. Then I apply multer middleware to the route using either upload.single('field') for a single file or upload.array('field', count) for multiple files.

After upload, the file metadata is available in req.file or req.files, and I can use that to store paths in a database or process the file as needed.

**11) How do you implement rate limiting or throttling in Express?**

To implement rate limiting in Express, I use the express-rate-limit middleware.

It allows me to control how many requests each client can make in a given time window. I typically configure it with a 15-minute window and limit each IP to a specific number of requests, say 100.

If a client exceeds that limit, they receive a 429 response. This helps protect the server from brute force attacks, abuse, or accidental overload.

For sensitive routes like login or APIs, I sometimes apply stricter limits. The configuration is simple and can be applied globally or to specific routes.

What are common performance bottlenecks in Express, and how do you fix them?

**Lack of Caching**:

* Not caching frequently accessed data, like API responses or database queries, can result in redundant, slow operations for every request.
* **Fix**: Implement **in-memory caching** (e.g., using **Redis** or **Node-cache**), and use **HTTP caching headers** for static content.

 **Blocking the Event Loop**:

* CPU-intensive tasks like image processing or complex calculations can block the event loop, slowing down the entire server.
* **Fix**: Use **worker threads** or **background queues** (e.g., **Bull** or **RabbitMQ**) to offload heavy computations.



**12) Can you explain the role of middleware in Express? What are the different types?**

The Middleware is the function that handle HTTP request, perform operations, and pass the control to next middleware.

**Middleware** in Express are functions that run between the **request** and **response** cycle. They have access to req, res, and next().  
Middleware is used for tasks like logging, authentication, parsing, error handling, etc.

**Types of middleware:**

1. **Application-level middleware** – Defined using app.use() or route-specific.
2. **Router-level middleware** – Attached to an instance of express.Router().
3. **Built-in middleware** – Like express.json() and express.static().
4. **Third-party middleware** – Like body-parser, cors, morgan.
5. **Error-handling middleware** – Functions with four arguments (err, req, res, next).

**13) What is difference between app.use() and app.get() ?**

app.use() is used to **mount middleware** that executes **for all incoming requests** that **match the specified path** (or **all paths** if no path is given), **regardless of the HTTP method** (GET, POST, etc.).

app.get() is used to define a **route handler** for **HTTP GET requests** on a specific path.  
It is typically used when you want to handle browser visits or API requests made via GET.

**How do you handle form data and JSON data in Express?**

Express handles incoming request data using **middleware**:

* For **JSON data**, use express.json()
* For **URL-encoded form data**, use express.urlencoded({ extended: true })

**14) What is the difference between route parameters and query parameters? Give an example?**

Route parameters are part of the URL path and are used to identify a specific resource, like /user/:id, which maps to req.params.id.  
Query parameters come after a ? in the URL and are used to send additional information like filters or searches, such as /search?q=nodejs, accessed via req.query.q.

**15) Does Express use threads? How does Node handle multiple requests concurrently?**

**No, Express itself does not use threads.**  
It's built on **Node.js**, which uses a **single-threaded, event-driven architecture** powered by **libuv**.

🔹 Node handles concurrency using:

* An **event loop** for non-blocking tasks (like I/O operations).
* A **thread pool** (via libuv) for offloading blocking operations (e.g., file system access, DNS lookups, crypto).

So, while your JavaScript code runs on a single thread, Node can handle **many concurrent requests** using the **event loop** and **non-blocking I/O**.

**16) You mentioned worker\_threads and libuv earlier — can you explain when you’d use each?**

Yes. **libuv** and **worker\_threads** both help Node.js with concurrency, but they serve different purposes:

| **Feature** | **libuv (Built-in in Node.js)** | **worker\_threads (Node module)** |
| --- | --- | --- |
| Use case | Handles **I/O-bound** tasks (network, disk) | Used for **CPU-bound** tasks (computation-heavy) |
| How it works | Uses internal **thread pool** for async I/O | Runs JS code in **separate threads** (Workers) |
| Blocking risk | Keeps event loop non-blocking | Prevents blocking main thread during heavy work |
| Example use | File reads, HTTP requests, DB queries | Image processing, encryption, data parsing, ML tasks |

**17)** **Imagine you're handling a CPU-heavy task like image processing or generating reports — how would you keep your Express app responsive?**

To keep Express responsive during CPU-heavy tasks, I would avoid running them directly in the main thread. Instead, I’d use worker\_threads for internal offloading or a background job queue like Bull for asynchronous processing. This ensures the event loop stays unblocked and the app can handle other requests smoothly.

**Common Node Modules (fs, http, path, etc.)**

**1. How does the fs module work? Explain sync vs async file read/write.**

The fs module in Node.js is used for interacting with the file system (read, write, update, delete files).

**🔹 Sync (Synchronous)**

* Blocks the execution of code until the operation completes.
* Slower for large files or in production environments.

const fs = require('fs');

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

console.log(data); // Will only run after the file is read

**🔹 Async (Asynchronous)**

* Non-blocking: Continues code execution while file I/O completes in the background.
* Recommended for performance and scalability.

const fs = require('fs');

fs.readFile('example.txt', 'utf8', (err, data) => {

if (err) throw err;

console.log(data); // Output after file read completes

});

**20. What is the difference between require and import in Node.js?**

| **Feature** | **require** | **import (ES Modules)** |
| --- | --- | --- |
| Syntax Type | CommonJS | ES6 Modules |
| When Available | Default in Node.js | Requires "type": "module" in package.json |
| Dynamic Import | Can be called anywhere | Must be at top-level (or use import() function) |
| File Extension | Often .js, .json | Must specify .js unless using bundler |

**How would you create a simple web server using only http?**

js

CopyEdit

const http = require('http');

const server = http.createServer((req, res) => {

res.writeHead(200, { 'Content-Type': 'text/plain' });

res.end('Hello World');

});

server.listen(3000, () => {

console.log('Server running at http://localhost:3000');

});

**Explain the difference between fs.readFile() and fs.createReadStream().**

| **Feature** | **fs.readFile()** | **fs.createReadStream()** |
| --- | --- | --- |
| Type | Asynchronous, reads full file | Stream-based, reads in chunks |
| Memory Usage | High for large files | Efficient and scalable |
| When to Use | Small files | Large files, continuous processing |

**Can you explain blocking code with a real-world analogy?**

**🧠 Real-world analogy:**

**Blocking Code:**  
Imagine you're at a restaurant and you order food. But instead of taking other orders, the waiter stands still and waits in the kitchen until your food is ready. Nobody else can be served — that’s blocking.

**Non-Blocking Code:**  
Now imagine the waiter takes your order, passes it to the kitchen, and continues serving other customers. When your food is ready, he brings it to you — that’s non-blocking (how Node.js is designed to work).

**What is the dns module in Node.js and why would you use it?**

The dns module allows you to perform DNS (Domain Name System) operations — such as resolving domain names to IP addresses, retrieving record types like A, MX, TXT, and performing reverse lookups.  
It's useful in applications where you need domain verification, server health checks, or custom DNS resolution.

**What's the difference between dns.lookup() and dns.resolve()?**

| **Feature** | **dns.lookup()** | **dns.resolve()** |
| --- | --- | --- |
| Uses system DNS? | ✅ Yes (OS-level resolver) | ❌ No (performs real DNS queries) |
| Async I/O? | Yes | Yes |
| Caching? | Depends on OS | No, always fresh |
| Use case | Resolve domain to IP | Query specific DNS record types |

**What is the purpose of dns.reverse()? Give a real-world use case.**

dns.reverse(ip, cb) performs a **reverse DNS lookup**, resolving an IP address to hostnames.