**1 - What is cors ? how to add headers ?why we use it?s**

A - CORS stands for Cross-Origin Resource Sharing. It is a security feature implemented by web browsers to restrict cross-origin HTTP requests initiated from scripts running in the browser. A cross-origin request occurs when a web application hosted at one origin (domain, protocol, or port) makes a request to a different origin.

**2 - What is RestFul API , how does it work?**

**A -** A RESTful API (REST API) is an application programming interface (API) that adheres to the principles of REST. It provides a standard way for software applications to communicate over the internet or a network by using standard HTTP methods such as GET, POST, PUT, DELETE, etc., and following RESTful principles.

Until 2000 SOAP of microsoft was used widely

Here's how REST APIs work

****Stateless Communication****: REST APIs are stateless, meaning that each request from a client to the server contains all the information needed to understand and process the request. The server does not store any client state between requests. This simplifies server implementation and enhances scalability and reliability.

****Resource-Based****: In a RESTful API, resources are the key abstraction. A resource is any information that can be accessed or manipulated by the client, such as a user, a product, or a document. Each resource is identified by a unique URI (Uniform Resource Identifier) or URL (Uniform Resource Locator).

****HTTP Methods****: RESTful APIs use standard HTTP methods (GET, POST, PUT, DELETE, etc.) to perform CRUD (Create, Read, Update, Delete) operations on resources. These HTTP methods correspond to the actions that can be performed on resources:

* + GET: Retrieve a resource or a collection of resources.
  + POST: Create a new resource.
  + PUT: Update an existing resource or create a new resource if it does not exist.
  + DELETE: Delete a resource.
  + PATCH: Partially update an existing resource.
  + OPTIONS: Retrieve information about the supported HTTP methods and headers for a resource.
  + HEAD: Retrieve only the headers for a resource without the body.

****Uniform Interface****: RESTful APIs use a uniform interface for communication between clients and servers. This includes standard HTTP methods, resource URIs, and representations of resources (e.g., JSON, XML).

****Representation of Resources****: Resources are represented using a format such as JSON (JavaScript Object Notation) or XML (eXtensible Markup Language). Clients and servers communicate by exchanging representations of resources, allowing different applications to understand and process the data.

****Hypermedia as the Engine of Application State (HATEOAS)****: HATEOAS is a constraint in RESTful APIs that allows clients to navigate the API dynamically by following hyperlinks embedded in resource representations. This enables clients to discover and interact with resources without prior knowledge of their URIs.

Overall, RESTful APIs provide a flexible and scalable approach to building web services that can be consumed by a wide range of clients, including web browsers, mobile devices, and other applications. They leverage standard HTTP protocols and utilize the principles of REST to provide a uniform and efficient way for clients and servers to communicate over the internet or a network.

窗体底端

**3 - How do you do error logging in your node js application**

**A -**  . error logging is essential for monitoring and troubleshooting. There are various ways to implement error logging, ranging from basic console logging to using dedicated logging libraries like Winston or Bunyan.

**4 - How do you do debugging**

A - Debugging Node.js applications typically involves using a combination of built-in debugging tools, logging, and third-party tools. Here's a high-level overview of the process:

1. \*\***Console.log**\*\*: The simplest form of debugging involves adding `console.log` statements strategically throughout your code to print out variable values, function calls, or any other relevant information.

2. **adding Debugger breakpoint** : You can insert the `debugger` statement in your code, which will cause Node.js to break execution at that point and give you a debugging prompt in your terminal if you run your code with the `--inspect` flag.

3. \*\***Node.js Inspector**\*\*: Node.js comes with a built-in inspector that allows you to debug your code using the Chrome DevTools interface. You can start your Node.js application in debug mode by running `node --inspect your-script.js`. Then, you can open `chrome://inspect` in Chrome and click on "Open dedicated DevTools for Node".

5. \*\***Logging Libraries**\*\*: There are several logging libraries available for Node.js, such as Winston or Bunyan, which allow you to log messages at different levels of severity (e.g., debug, info, error) and direct those logs to different destinations

6. \*\***Third-Party Debugging Tools**\*\*: There are also third-party debugging tools and services available, such as New Relic or **Sentry**, which provide advanced monitoring, debugging, and error tracking capabilities for Node.js applications.

**5 - Explain everything abt JWT token ?**

JWT (JSON Web Token) is a compact, URL-safe way of representing claims to be transferred between parties. It consists of three parts:

1. \*\***Header**\*\*: Contains the type of token and the signing algorithm.

2. \*\***Payload**\*\*: Contains claims (statements) about an entity and additional data.

3. \*\***Signature**\*\*: Used to verify that the sender of the JWT is who it says it is and to ensure the message wasn't changed.

They are self-contained and stateless, making them efficient for distributed systems. However, ensure proper handling and security practices, such as using strong cryptographic algorithms and securely storing keys.

**IMP**

**6 - What is token and session authentication**

**A -**

**Session-Based Authentication:**

**\*\*1. Overview:\*\***

- In session-based authentication, when a user logs in, the server creates a session for that user.

- The session typically includes a unique session identifier (session ID) and any relevant user data (e.g., user ID, username).

- The **session ID** is usually stored in a cookie on the client side and sent with each subsequent request to the server.

- The server validates the session ID and retrieves the associated user data to authenticate the user and authorize access to protected resources.

\*\*2. Process:\*\*

- \*\*Login\*\*: When a user logs in with their credentials, the server authenticates the user's credentials.

- \*\*Session Creation\*\*: If the credentials are valid, the server creates a session for the user and stores session data on the server.

- \*\*Session ID\*\*: The server sends the session ID to the client, typically in a cookie.

- \*\*Subsequent Requests\*\*: The client includes the session ID in each request to the server.

- \*\*Server Validation\*\*: Upon receiving a request, the server validates the session ID.

- \*\*Access Control\*\*: If the session is valid, the server retrieves the associated user data from the session and grants access to the requested resource.

**Session Storage Options:**

In-Memory Storage: Sessions can be stored directly in memory on the server. While this is fast, it's not suitable for distributed systems and can lead to issues with scalability and reliability. Database Storage: Session data can be stored in a database, such as MongoDB or Redis.

This allows for scalability and persistence across server restarts. External Session Stores: Services like Redis or Memcached can be used as external session stores, providing fast and scalable storage specifically designed for session management.

\*\*3. Key Points:\*\*

- Sessions are stored on the server, which can make scaling more challenging.

- Sessions are typically managed using server-side technologies such as Express.js (with Express Session middleware) in Node.js applications.

- Session-based authentication is vulnerable to certain attacks such as session hijacking or session fixation if proper security measures are not implemented.

### Token-Based Authentication:

\*\*1. Overview:\*\*

- In token-based authentication, when a user logs in, the server issues a token to the client.

- The token contains relevant user information (claims) and is digitally signed by the server to ensure its integrity.

- The token is then sent by the client with each request to the server to authenticate and authorize access.

\*\*2. Process:\*\*

- \*\*Login\*\*: When a user logs in, the server authenticates the user's credentials.

- \*\*Token Issuance\*\*: If the credentials are valid, the server generates a token containing relevant user information.

- \*\*Token Delivery\*\*: The server sends the token to the client, typically in the response body or as a response header.

- \*\*Subsequent Requests\*\*: The client includes the token in the Authorization header of each request to the server.

- \*\*Token Validation\*\*: Upon receiving a request, the server verifies the token's signature to ensure its authenticity and integrity.

- \*\*Access Control\*\*: If the token is valid, the server extracts the user information from the token and grants access to the requested resource.

\*\*3. Key Points:\*\*

- Tokens are typically JWTs (JSON Web Tokens) in modern applications.

- Tokens are stateless, meaning the server does not need to store session data, making scaling easier.

- Token-based authentication is commonly used in distributed systems and APIs.

- Token expiration and token revocation mechanisms should be implemented to enhance security.

- Token-based authentication is susceptible to certain attacks such as token theft if not implemented securely.

### Comparison:

- \*\*Storage\*\*: Sessions are stored on the server, while tokens are typically stored on the client side.

- \*\*Scalability\*\*: Token-based authentication is more scalable due to its stateless nature.

- \*\*Security\*\*: Both methods require proper implementation to ensure security, including measures such as encryption, token expiration, and protection against common attacks.

- \*\*Use Case\*\*: Session-based authentication is commonly used in traditional web applications, while token-based authentication is preferred in modern APIs and microservices architectures.

Both session-based and token-based authentication have their pros and cons, and the choice between them depends on factors such as the application's requirements, architecture, and security considerations.

7 -