**Interview Question**

**Question 1**

**https://medium.com/swlh/all-you-need-to-know-about-json-web-token-jwt-8a5d6131157f**

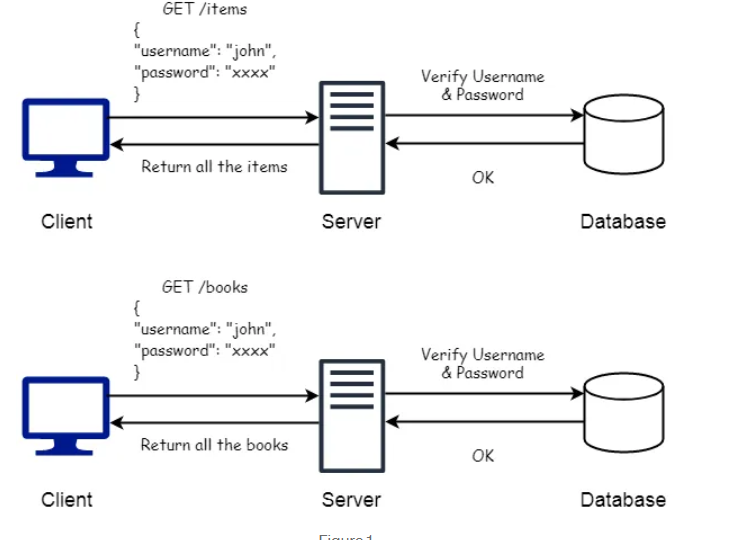
**https://medium.com/@extio/understanding-json-web-tokens-jwt-a-secure-approach-to-web-authentication-f551e8d66deb**

**Implement security using token**

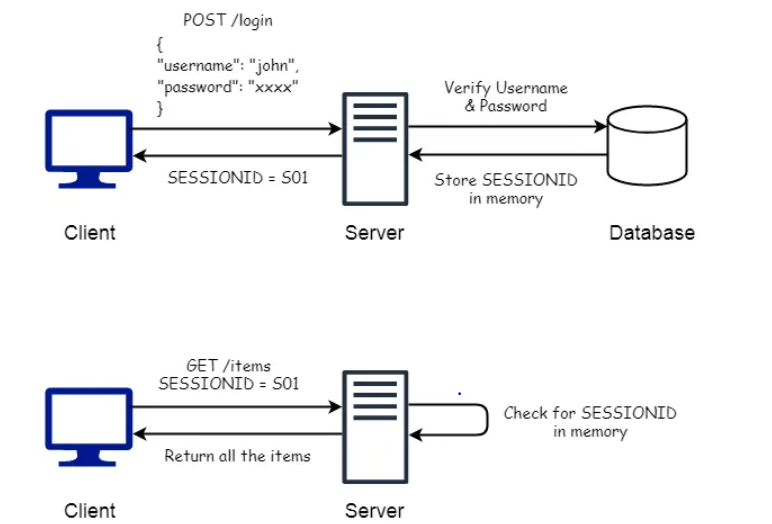
*“A JSON Web Token (JWT), pronounced ‘jot’, is an open standard (*[*RFC 7519*](https://tools.ietf.org/html/rfc7519)*) which is used for securely transmitting information between entities as a JSON object.”*

**Why we need JSON Web Token?**

HTTP is a stateless protocol that means a new request does not remember anything about the previous one. So for each request, you need to login and authenticate yourself (figure 1). Now, this sounds like a lot of work.

****

So, the solution to deal with this is the use of what’s called a session. A session is an object stored on the server that helps the user to stay logged in or to save any reference to their account. Figure 2 shows the overall flow of this process.

****

## Problems with Session-based Authentication

**Scalability :**The sessions need to be stored on the server, either in memory or in a database. In the current API era, a server can receive a large number of requests, and hence the server needs to be scaled up. Adding new resources can increase the complexity of the infrastructure as well.

**Session Management:**The server needs to manage the sessions by keeping track of all the active, inactive (expired, invalidated) sessions. The expired or invalidated sessions need to be removed from the memory.

**Performance:**For every request, the server needs to perform a lookup into the memory to check if the provided session object is valid or not. This back and forth can mark down the server.

**The better and effective solution**

The JSON Web Token (JWT) does not use sessions and hence prevents the above problems. When you send your credentials to the server instead of making a session, the server will return a JSON Web Token. You can use that JWT to do whatever you want with the server (Of course, the things that you are authorized to do).

Consider a JWT like a hotel key: When you enter the hotel, first you need to register yourself at the reception to receive your key card. You can use that key card to open and close your room, access common amenities like Bar, Fitness Centre, etc. But you cannot use that key card to access someone else’s room or Manager’s office since you are not authorized to do so. The key card comes with an expiration date, and it becomes useless once your stay has ended at the hotel.

**Similarly, you can use your JWT token generated from one server to access resources on different servers.** The JWT token contains claims like expiration date/time that can be used to check its validity..

**A JWT consists of three parts separated by periods (.), which are base64url-encoded strings:**

1. **Header**: The header typically consists of two parts — the token type (JWT) and the signing algorithm being used, such as HMAC SHA256 or RSA.

Example:



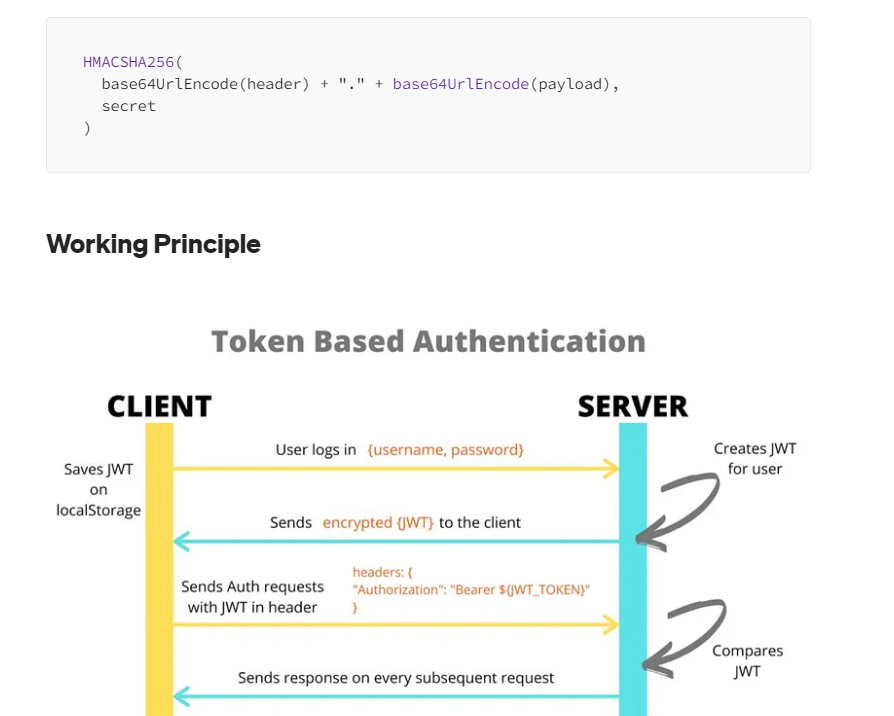
2. **Payload**: The payload contains the claims, which are statements about the user or other data. Claims can be of three types: registered, public, and private claims.

Example:

****

3. **Signature**: To create the signature part, you need to take the encoded header, encoded payload, a secret, and the algorithm specified in the header, then sign that with the secret. The signature is used to verify that the sender of the JWT is who it says it is and to ensure that the message wasn’t changed along the way.

Example (using HMAC SHA256):

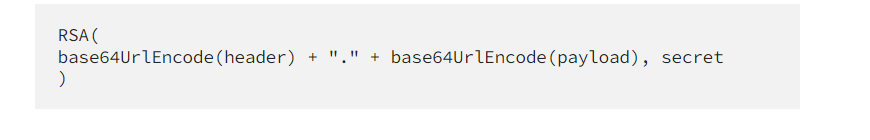
****

## Signature

Signature is the most important part of JWT which helps to verify if the information within the token has been tampered with or not. It can be also used to verify that the sender of the JWT is who it says it is.

In order to calculate the signature, you require three things: an encoded header, an encoded payload, and a secret. First, you will take the encoded header and encoded payload and concatenate them with a period separator to form a string. This concatenated string will be hashed using an algorithm specified in the header and a secret key to calculate the signature.

Consider the following example where we will be using the RSA algorithm to generate a signature.

****

Now let’s put all three parts together to get a JSON Web Token.

We have three Base64-URL strings, which can be concatenated with a period between each. The JWT is very compact and can be easily exchanged in HTML and HTTP environments.

*The header and payload can be easily decoded (since it’s just base64) to retrieve information contained within the token. The signature can be just used to maintain the integrity of the token and not to secure the contained information. Therefore, you should pass any sensitive information in the token and if you want to pass make sure you encrypt the token to secure it.*

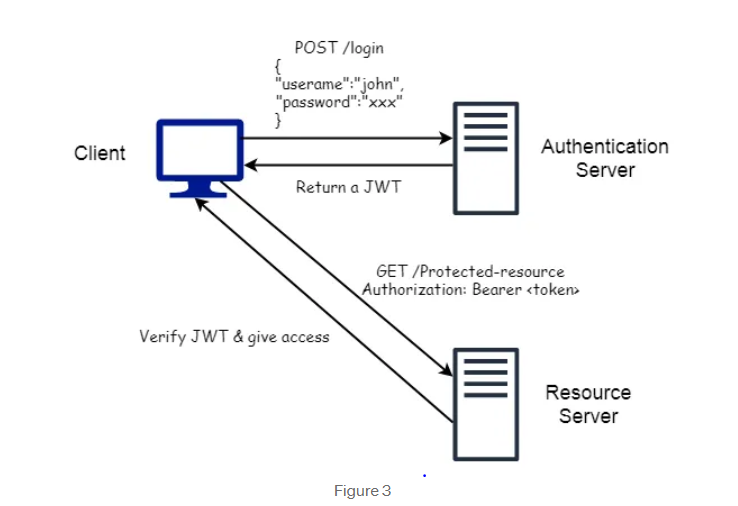
**How does a JSON Web Token work?**

When a user sends his credentials to the server to login, the server authenticates the user. If the authorization is successful, the server sends a JSON Web Token to the user. The user can use the JWT to request any protected services/resources from the server by including the JWT in the **Authorization** header using the **Bearer** schema.

Authorization: Bearer <token>

When the server gets a request from the user to access any protected content, the protected routes of the server will look for a valid JWT in the Authorization header. If the token is present and is valid the server will allow access to the user.

The JWT contains necessary information about the user that can be used to identify the user, know the user’s privilege, and serve the user accordingly. Because of JWT, the server does not need to query the database every time a request comes in to check if the user has the necessary rights or not.

****

Every token assigned by the server is signed by a secret key known to the server only. Therefore, only the server can use the secret key to verify the token and to check if the token has tampered. If an attacker tries to make any changes in the token (like granting admin privileges), the signature of the token needs to be calculated again that will require the secret key. Since the attacker does not have a secret key making any changes to the token will invalid it. The server will discard such requests to prevent unauthorized access.

**Advantages of JSON Web Token**

* **Compactness:**JSON is less verbose than XML and therefore when it is encoded it takes up less space making JWT more compact then SAML.
* **No need of Session:**The JWT can contain all the necessary information about the user and therefore there is no need to maintain a session object on the server, saving up server memory.
* **Built-in Expiration:** The JWT has claims that can be used to assign it a expiration date/time. Therefore, the token can become invalid on its own after the expiration period.
* **No need of Cookies:**The token can be stored in the localStorage, indexDB, or some native store. This will provide a protection against CORS and CSRF attacks.
* **Compatibility:**In most programming languages, JSON parsers are popular because they map directly to objects. Contrary, there is no natural document-to-object mapping in XML. This makes it simpler than SAML assertions to operate with JWT.

Question 2

when we use restcontroller what annotation we will use

* **@RestController**: This annotation indicates that the class is a REST controller. It combines **@Controller** and **@ResponseBody**, meaning that each method will return the response body directly, rather than relying on a view resolver to render a web page.
* **@RequestMapping("/hello")**: This annotation maps the method **hello()** to the **/hello** endpoint. When a request is made to **/hello**, the **hello()** method will be invoked, and its return value will be sent as the response.

So, when you use **@RestController**, Spring Boot automatically serializes the return value of your methods into JSON (by default) and sends it as the response. This is very convenient for building RESTful APIs.

Question 3

How will you maintain version of api both old and new

Maintaining both old and new versions of an API is a common requirement, especially in large-scale applications where backward compatibility is important. Here's a strategy for managing multiple versions of an API:

### 1. ****Versioning in the URI****:

One common approach is to include the version number in the URI of the API endpoints. For example:

* **/api/v1/resource**
* **/api/v2/resource**

### 2. ****Using Custom Headers****:

Another approach is to use custom headers to specify the API version. Clients can include the version in the request headers, and the server can route the request accordingly.

### 3. ****URL Parameters****:

You can also use URL parameters to specify the API version. For example:

* **/api/resource?version=1**
* **/api/resource?version=2**

### 4. ****Content Negotiation****:

Content negotiation allows clients to request a specific version of the API by specifying the version in the **Accept** header of the HTTP request.

Question 4 :----

300 series

HTTP 3xx status codes indicate redirection responses in HTTP. These status codes inform the client that the requested resource is located elsewhere and provide information on how the client can retrieve it. Here are some common status codes in the 3xx series:

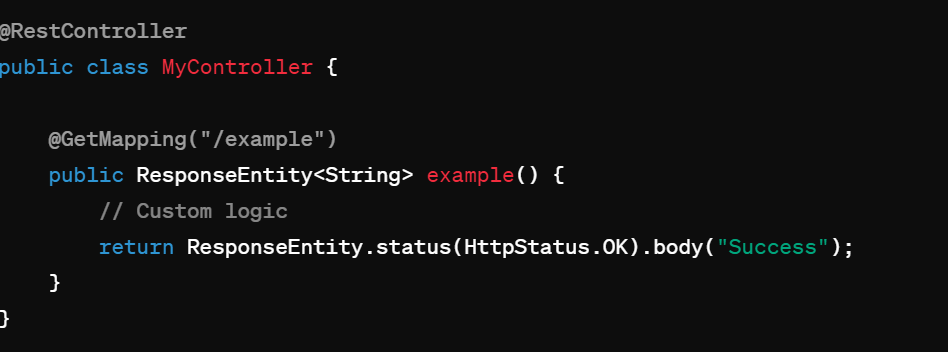
1. **301 Moved Permanently**: This status code indicates that the requested resource has been permanently moved to a new location. The client should update its bookmarks or links to point to the new URL provided in the response.
2. **302 Found**: This status code indicates that the requested resource temporarily resides at a different location. The client should use the new URL provided in the response for the current request, but future requests may still use the original URL.
3. **303 See Other**: This status code indicates that the client should retrieve the resource using a GET request to the URL provided in the response's **Location** header. It is often used in response to a POST request to redirect the client to a different URL.
4. **307 Temporary Redirect**: Similar to 302, this status code indicates that the requested resource temporarily resides at a different location. However, the client should use the same HTTP method (e.g., GET, POST) for the redirected request.

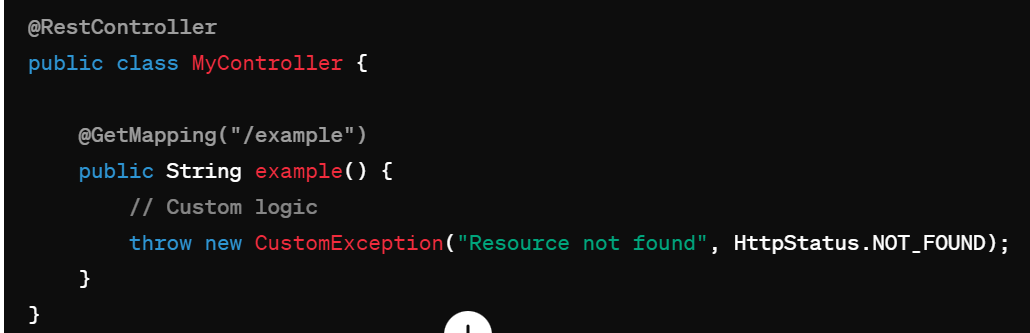
Question 5

How to implement response status ?

In Spring Boot, you can easily set the HTTP response status code using various methods provided by the **ResponseEntity** class or by throwing specific exceptions. Here's how you can implement response codes in Spring Boot:

### Using ResponseEntity:





Send 500



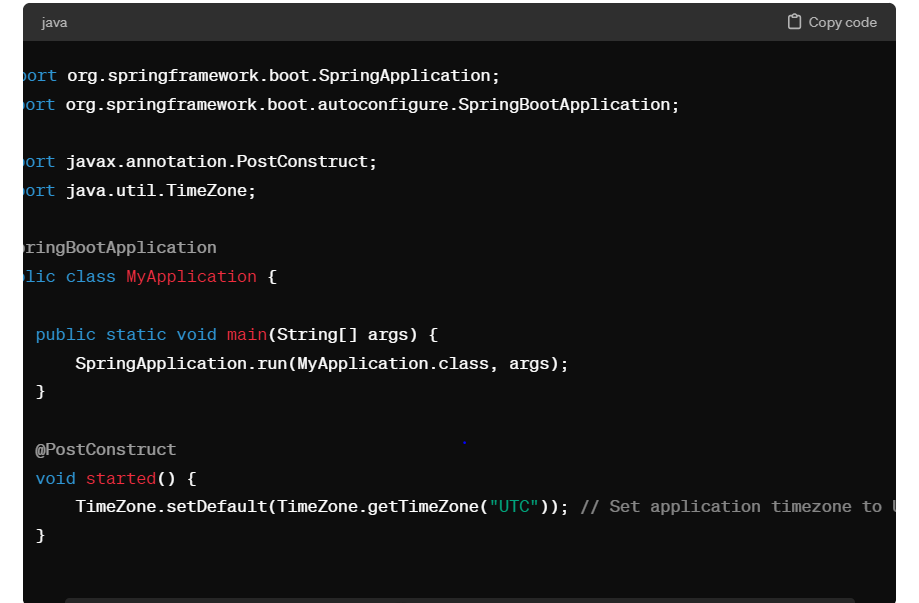
Question 6

**how to manage timezone of application and database present in different location using springboot**

When your application and database are located in different timezones, it's crucial to ensure that they are synchronized to handle time-related operations accurately. Here's how you can manage timezones effectively in a Spring Boot application with a database located in a different location:

### 1. Configure Timezone in Spring Boot Application:

In your Spring Boot application, you can set the default timezone in the main application class.



### 2. Configure Timezone in Database:

Ensure that your database server is configured to use the UTC timezone. The steps to configure timezone settings may vary depending on the database you are using. For example, in MySQL, you can set the timezone using:

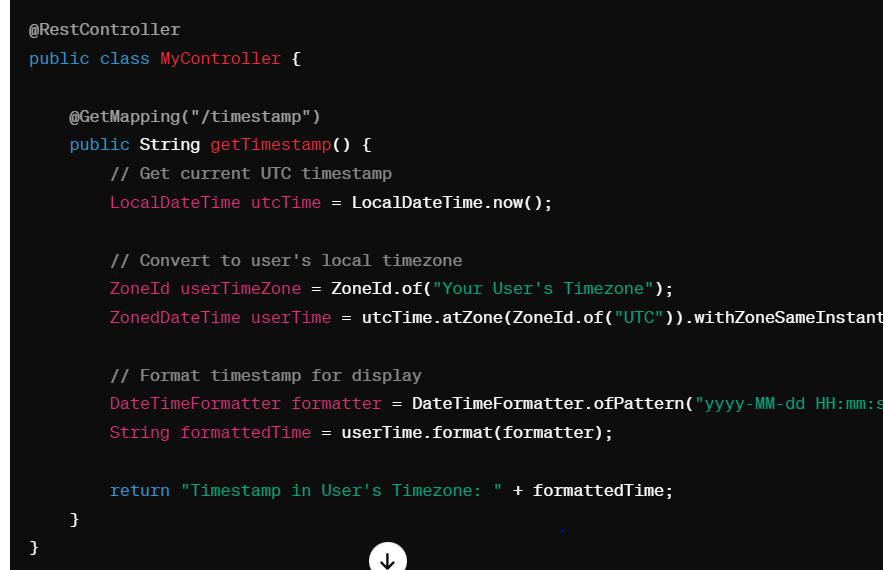


### 3. Store Timestamps in UTC:

When storing timestamps in the database, use UTC timezone to maintain consistency.

### 4. Convert Timezones for Display:

In your Spring Boot controllers or services, convert timestamps from UTC to the user's local timezone before presenting them to the user.



### 5. Test Across Timezones:

Thoroughly test your Spring Boot application across different timezones to ensure that date and time-related functionality behaves as expected in various scenarios.

By following these steps, you can effectively manage timezones in your Spring Boot application with a database located in a different location, ensuring consistency and accuracy in date and time-related operations across different geographical regions. Adjust the code according to your specific requirements and database configuration.

Question 7

Difference between @Controller and @RestController

n Spring MVC, both **@Controller** and **@RestController** annotations are used to define controller classes, but they serve different purposes and have different behaviors:

1. **@Controller:**
   * The **@Controller** annotation is used to define a controller class in Spring MVC.
   * Controllers annotated with **@Controller** are typically used to handle traditional web requests and return views.
   * Methods in a **@Controller** class can return a logical view name, which is resolved by the view resolver to render the actual view (e.g., JSP, Thymeleaf, Freemarker).
   * Controllers annotated with **@Controller** are suitable for building applications that render HTML views and follow the MVC (Model-View-Controller) architectural pattern.
   * Example:

**@RestController:**

* The **@RestController** annotation is a specialized version of the **@Controller** annotation introduced in Spring 4.
* Controllers annotated with **@RestController** are used to build RESTful web services that return data in JSON or XML format.
* Unlike **@Controller**, methods in a **@RestController** class directly return domain objects or collections, which are automatically serialized into JSON or XML by Spring's **HttpMessageConverters**.
* **@RestController** eliminates the need for annotations like **@ResponseBody** because it combines **@Controller** and **@ResponseBody**.

**In summary, the main difference between @Controller and @RestController is their intended use case:**

* Use **@Controller** for building web applications that render HTML views and follow the MVC pattern.
* Use **@RestController** for building RESTful web services that return data in JSON or XML format without rendering views.

Questions 8:

Bean scopes

In Spring Framework, bean scope determines the lifecycle and visibility of a bean instance. Different scopes are available to control how Spring manages and creates bean instances. Here are the common bean scopes in Spring:

1. **Singleton:**
   * Default scope in Spring.
   * Only one instance of the bean is created per Spring IoC container.
   * All requests for the bean result in the same shared instance.

**Prototype:**

* A new instance of the bean is created every time it is requested.
* Multiple instances of the bean can coexist within the Spring IoC container.
* The lifecycle of a prototype bean is not managed by Spring; it is the responsibility of the application to manage the bean's lifecycle.

**Request:**

* A new instance of the bean is created for every HTTP request.
* This scope is only available in a web-aware Spring application context (e.g., web applications using Spring MVC).
* The bean instance is available only within the scope of the current HTTP request.

**Session:**

* A new instance of the bean is created for every HTTP session.
* Similar to request scope, but the bean instance is available throughout the entire HTTP session.
* This scope is only available in a web-aware Spring application context.

**Application/Singleton (deprecated):**

* Deprecated since Spring 3.0. Use Singleton scope instead.
* Similar to Singleton scope, but scoped at the level of the entire web application.
* This scope is only available in a web-aware Spring application context.

**WebSocket:**

* Introduced in Spring 4.2 for use with WebSocket-based applications.
* A new instance of the bean is created for every WebSocket session.
* This scope is only available in a web-aware Spring application context.

Question 9:

Define ioc container :-

Inversion of Control (IoC) container is a core component of the Spring Framework that manages the creation, configuration, and lifecycle of application objects (beans). The IoC container is responsible for removing the responsibility of object creation and dependency injection from the application code, thereby promoting loose coupling and improving testability and maintainability.

Here's a more detailed explanation of what an IoC container does:

1. **Object Creation:**
   * The IoC container creates and manages instances of application objects (beans) based on the configuration provided by the developer.
   * Developers define beans by annotating classes with Spring annotations such as **@Component**, **@Service**, **@Repository**, or **@Controller**.
2. **Dependency Injection:**
   * The IoC container injects dependencies into the beans during instantiation, thereby removing the need for the beans to create or lookup dependencies themselves.
   * Dependencies can be injected through constructor injection, setter injection, or field injection, depending on the configuration.
3. **Lifecycle Management:**
   * The IoC container manages the lifecycle of beans, including instantiation, initialization, and destruction.
   * Initialization and destruction callbacks can be defined using annotations such as **@PostConstruct** and **@PreDestroy**.
4. **Configuration Management:**
   * The IoC container allows developers to configure beans and their dependencies using various configuration mechanisms, such as XML configuration, Java-based configuration, or annotations.
   * Configuration metadata specifies how beans are created, wired together, and managed by the IoC container.
5. **Scoping:**
   * The IoC container supports different bean scopes, such as singleton, prototype, request, session, etc., to control the lifecycle and visibility of beans.
   * Developers can specify the scope of a bean using annotations or XML configuration.

By providing these features, the IoC container enables developers to focus on implementing business logic without worrying about managing object creation, wiring dependencies, or handling bean lifecycles. The IoC container promotes the principle of dependency inversion and facilitates the development of modular, loosely coupled, and testable applications.

Question 10

What is spring boot actuator

Spring Boot Actuator is a sub-project of Spring Boot that provides production-ready features to help monitor and manage Spring Boot applications. It exposes various HTTP endpoints and provides valuable insights into the internals of your application at runtime. Actuator endpoints can be used for monitoring, health checks, metrics collection, auditing, and more.

Here are some key features of Spring Boot Actuator:

1. **Health Checks:**
   * The **/actuator/health** endpoint provides information about the health of the application and its dependencies.
   * Health indicators can be customized to include checks for databases, message brokers, external services, etc.
   * Monitoring tools can use this endpoint to check the overall health status of the application.
2. **Metrics:**
   * The **/actuator/metrics** endpoint provides detailed metrics about various aspects of the application, such as memory usage, thread pool utilization, garbage collection stats, HTTP request metrics, etc.
   * Metrics can be exported to monitoring systems like Prometheus, Grafana, or Micrometer for analysis and visualization.
3. **Information Endpoint:**
   * The **/actuator/info** endpoint exposes arbitrary application information.
   * Developers can provide custom metadata about the application, such as version, description, or any other relevant details.
4. **Environment Endpoint:**
   * The **/actuator/env** endpoint exposes information about the application's environment properties.
   * It allows you to inspect the current configuration of the application, including properties from the application.properties/yml file, system properties, and environment variables.
5. **Beans Endpoint:**
   * The **/actuator/beans** endpoint provides a complete list of all Spring beans in the application context.
   * It is useful for debugging and understanding the bean wiring in the application.
6. **Auditing:**
   * Spring Boot Actuator provides features for auditing HTTP requests, including information such as who made the request, when it was made, and the outcome of the request.

Question 11

How to establish connection between two microservices ?

# 

# 2 types communication in m/s :----

# Synchronous blocking

# Asynchronous non-blocking

# 

# 

**Asynchronous non blocking :**  the microservice send some kind of data or events or request to another microservice and does not wait for response and it will not block any of the operation in the meantime

