1) What are the pros and cons of using microservice architecture?

Pros:

- **Scalability**: Microservices allow independent scaling of components based on demand.
- **Flexibility in Technology**: You can use different technologies for different services, making the system flexible.
- **Resilience**: Failure in one service does not affect others due to isolation.
- **Faster Deployment**: Since services are independent, teams can deploy their services separately without affecting others.
- **Improved Maintenance**: Smaller codebases are easier to manage and maintain.

Cons:

- **Complexity**: Managing multiple services increases system complexity, especially in terms of inter-service communication.
- **Data Consistency**: Distributed systems often face challenges with data consistency, particularly with eventual consistency.
- **Deployment Overhead**: Microservices require sophisticated deployment tools and continuous integration/continuous delivery (CI/CD) pipelines.
- **Increased Latency**: Communication between services can increase latency, especially if services are deployed across different servers or regions.
- **Testing Challenges**: Testing interactions between microservices can be more complex than testing monolithic applications.

---

2) Why do you use microservice architecture?

Microservice architecture is used to:

- **Break down complex applications** into smaller, manageable services.
- **Enable independent development and deployment** of services by different teams, allowing faster time-to-market.
- **Improve fault tolerance** by isolating failures to individual services, reducing the risk of a system-wide outage.
- **Support scaling** specific parts of the application based on demand, rather than scaling the entire application.

---

3) Why do you use Eureka Server?

Eureka Server is used in microservice architectures for **service discovery**. It allows:

- **Automatic registration and deregistration** of services.

- **Dynamic discovery** of services at runtime, enabling microservices to locate and interact with each other.
- **Load balancing** by routing requests to the correct instance of a service.
- **Fault tolerance** by providing failover capabilities when a service instance is unavailable.

---

4) Why is there a requirement of using Solace in the architecture?

Solace is used for **messaging and event streaming** in architectures, especially for real-time communication. The reasons for using Solace include:

- **Asynchronous communication**: Allows decoupling of services by providing an event-driven, message-based architecture.
- **Scalability and high availability**: Solace handles high message throughput and provides fault tolerance.
- **Efficient message routing**: Solace's publish-subscribe model ensures messages are routed to the appropriate consumers.
- **Integration**: Solace can integrate with various systems (like IoT, cloud, and enterprise systems) using standard protocols.

---

5) Why do we write server.port=0 in application.properties?

Setting server.port=0 in the application.properties file tells Spring Boot to **assign a random available port** when starting up the application. This is useful in situations like:

- **Test environments**, where you don't want a fixed port to be used.
- **Multiple instances of the same application** running on different ports for load balancing or scaling purposes.

---

6) What is the single point of failure in your system design, and what will you do to mitigate it?

A **single point of failure** (SPOF) refers to any component in the system where failure would lead to a complete system failure. Common SPOFs include:

- **Database**: If a single database server fails, the entire system might go down.
- **Application server**: If a single server hosting the application fails, it might affect availability.

Mitigation strategies:

- **Load balancing** to distribute traffic across multiple instances.
- **Replication** of databases across different nodes or regions.
- **Failover mechanisms** to switch to backup servers in case of failure.
- **Distributed architecture** to ensure redundancy and fault tolerance.

---

7) What is database partitioning and indexing logic?

- **Database partitioning** involves dividing large databases into smaller, more manageable pieces (partitions) based on certain criteria (e.g., range, hash, list). This improves performance and scalability.
    - Example: Splitting a table by date (e.g., monthly partitions).
- **Indexing** improves database query performance by creating data structures (indexes) that speed up search operations.
    - **B-tree indexes**: Common for equality and range queries.
    - **Hash indexes**: Optimized for equality lookups.
    - **Full-text indexes**: Used for text searching.

---

8) How does Ansible Tower help for deployment process?

Ansible Tower provides an easy-to-use interface for managing **Ansible playbooks** and automating deployment tasks:

- **Centralized control**: It allows you to run and schedule playbooks across multiple servers from a single location.
- **Job templates**: Define reusable deployment processes.
- **Inventory management**: Keep track of servers and their configurations.
- **Logging and auditing**: Track deployment activities with logs for accountability and troubleshooting.
- **Role-based access control**: Ensure that only authorized users can trigger deployments.

---

9) How did you define your swagger.json for all the APIs that you have created?

Swagger documentation for APIs is typically defined using **annotations** in your Spring Boot controllers.

- **@Api**: Defines the class for Swagger documentation.
- **@ApiOperation**: Describes a specific API operation.
- **@ApiParam**: Describes parameters for an operation.
- You can use tools like **Springfox** or **Springdoc OpenAPI** to generate the swagger.json file automatically from your code annotations.

---

10) What is the topic selector concept in Solace and how does it make the message sending between publisher and consumer efficient?

The **topic selector** in Solace allows consumers to specify which messages they are interested in by filtering messages based on topic names and message properties. This makes message sending more efficient because:

- **Reduced message overhead**: Only relevant messages are sent to consumers.
- **Optimized resource usage**: Solace can deliver messages to specific consumers, reducing unnecessary message routing.
- **Real-time filtering**: Consumers can dynamically adjust their message filters based on changing requirements.

---

11) What is the annotation @SpringBootApplication consist of?

@SpringBootApplication is a composite annotation that combines:

- **@Configuration**: Marks the class as a source of bean definitions.
- **@EnableAutoConfiguration**: Enables Spring Boot's auto-configuration feature.
- **@ComponentScan**: Scans the current package and its sub-packages for Spring components (like controllers, services, etc.).

---

12) How does a Spring Boot project boot up?

When a Spring Boot application starts:

1. The **main method** is invoked, which triggers the SpringApplication.run() method.
2. The **Spring context** is created, and all the beans are initialized.
3. **Auto-configuration** is applied based on the environment and dependencies.
4. **Embedded server** (like Tomcat or Jetty) is started if the application is a web application.

---

13) How many types of injections are there in Spring Boot project?

There are three types of dependency injection in Spring:

1. **Constructor Injection**: Dependencies are provided through the class constructor.
2. **Setter Injection**: Dependencies are provided via setter methods.
3. **Field Injection**: Dependencies are injected directly into fields using the @Autowired annotation.

---

14) What all are the status codes that you define for APIs?

Common HTTP status codes include:

- **200 OK**: The request was successful.

- **201 Created**: A resource was successfully created.
- **400 Bad Request**: The request was invalid or malformed.
- **401 Unauthorized**: Authentication is required or failed.
- **403 Forbidden**: The server understands the request, but the client is not authorized.
- **404 Not Found**: The requested resource could not be found.
- **500 Internal Server Error**: A server error occurred while processing the request.

---

15) What is the difference between Spring Boot bean creation using ApplicationContext and BeanFactory method?

- **ApplicationContext**: It is a more feature-rich container and is used in production environments. It supports features like event propagation, AOP, and more. It pre-initializes all beans during the startup.
- **BeanFactory**: It is a more lightweight container and is used in resource-constrained environments. It lazily initializes beans, meaning that beans are created only when needed.

---

16) How do you handle exceptions in the whole Spring Boot project globally?

Spring Boot provides a global exception handling mechanism using the @ControllerAdvice annotation:

- **@ControllerAdvice**: A class annotated with this can handle exceptions globally across all controllers.
- **@ExceptionHandler**: Methods in the @ControllerAdvice class can handle specific exceptions.
- **ResponseEntityExceptionHandler**: It can be used to handle standard Spring MVC exceptions.

---

17) How does RestController work? Give an example for POST and GET API using RestController class.

@RestController is a specialized version of @Controller used to define RESTful APIs. It combines @Controller and @ResponseBody to simplify code.

Example:

java

Copy

@RestController

@RequestMapping("/api")

public class MyController {

```
    @GetMapping("/hello")
    public String getHello() {
        return "Hello, World!";
    }

    @PostMapping("/create")
    public ResponseEntity<String> create(@RequestBody MyObject obj) {
        // Logic to save the object
        return ResponseEntity.status(HttpStatus.CREATED).body("Object Created");
    }
}
```

---

18) What is ORM? How can you use it in your project?

ORM (Object-Relational Mapping) is a technique that allows developers to map Java objects to relational database tables. It simplifies database interactions by abstracting SQL queries.

Spring Boot uses **JPA (Java Persistence API)** for ORM:

- Use **@Entity** annotation to define a JPA entity.
- **@Repository** interface to interact with the database.
- Use **Spring Data JPA** to create repository methods without writing SQL.

Example:

java

Copy

```
@Entity
public class Employee {
    @Id
    private Long id;
    private String name;
    // Getters and setters
}

@Repository
```

```
public interface EmployeeRepository extends JpaRepository<Employee, Long> {

}
```