Java, Spring Boot, Microservices, and Angular Interview Preparation Guide

August 2025

Full Notes available in Telegram
Link in Bio

Contents

1 Core Java & OOPs **4**
1.1 Difference between String, StringBuilder, and StringBuffer . . . 4
1.2 Difference between .equals() method and == operator . . . 5
1.3 Difference between method overloading and method overriding . . . 5
1.4 What is Singleton class? . . . 5
1.5 Create Singleton class . . . 5
1.6 What is thread safety and how do you ensure thread-safe classes? . . . 6
1.7 How does HashMap work internally? . . . 6
1.8 Difference: HashMap vs Hashtable vs ConcurrentHashMap . . . 6
1.9 How does HashMap work with Employee object as key? . . . 6
1.10 What is immutability and how does it help in concurrency? . . . 7
1.11 What is volatile and synchronized? . . . 7
1.12 How many ways can an object be created in Java? . . . 7
1.13 What is the use of ResponseEntity? . . . 8
1.14 What is meant by functional interfaces? . . . 8
1.15 How do functional interfaces work? . . . 8
1.16 What is dependency injection and its types? . . . 8
1.17 Difference between IOC and Dependency Injection . . . 9
1.18 How many ways to achieve dependency injection and which is best? . . . 9
1.19 @Primary vs @Qualifier — which takes priority? . . . 9
1.20 How to create beans manually? . . . 9
1.21 What is the difference between @RestController and @Controller? . . . 9
1.22 What is @SpringBootApplication annotation? . . . 10
1.23 What are stereotype annotations in Spring Boot? . . . 10
1.24 What is path variable? . . . 10
1.25 What is the default server in Spring Boot? . . . 10
1.26 What is the default port in Spring Boot? . . . 10

2 Java 8 Features **11**
2.1 Java 8 features used in your project . . . 11
2.2 What is a lambda expression? . . . 11
2.3 Write code using lambda expression . . . 11
2.4 What is Stream API? . . . 11
2.5 Intermediate vs terminal operations in streams . . . 12
2.6 List of intermediate and terminal methods . . . 12
2.7 Why Java introduced default and static methods in interfaces . . . 12
2.8 How to create an immutable class in Java . . . 12

2.9 How to use groupingBy() in streams . . . 13
2.10 How to create an Optional of an employee object . . . 13
2.11 Use Java 8 streams to remove duplicates from a list . . . 13
2.12 Find 3rd highest salary using Java 8 streams . . . 13

3 Spring Boot & Microservices **14**
3.1 How do you configure different environments in Spring Boot? . . . 14
3.2 What is a Spring Boot profile and how did you use it? . . . 14
3.3 What is exception handling and what is an advisor? . . . 14
3.4 How does @Transactional annotation work? . . . 14
3.5 What are the design patterns used in microservices? . . . 15
3.6 Explain each design pattern and when to use them . . . 15
3.7 What is microservices architecture? . . . 15
3.8 Why microservices? . . . 15
3.9 How do microservices communicate with each other? . . . 15
3.10 Difference between synchronous and asynchronous communication . . . 15
3.11 When to use synchronous vs asynchronous communication . . . 16
3.12 What is Kafka and how have you implemented it? . . . 16
3.13 How have you used Spring Security in your project? . . . 16
3.14 What is JWT security and how have you used it? . . . 16
3.15 What are the things to consider while developing REST APIs? . . . 17
3.16 Difference between PUT, POST, and PATCH . . . 17
3.17 API versioning strategies . . . 17
3.18 What is idempotency and why is it critical? . . . 17
3.19 How do you achieve idempotency in microservices? . . . 18
3.20 Securing APIs using JWT and OAuth2 . . . 18
3.21 Validating incoming payloads with annotations . . . 18
3.22 What are the steps to test Spring Boot microservices applications? . . . 18
3.23 How to handle exceptions in Spring Boot . . . 19
3.24 How to create global exceptions and what annotations are used . . . 19
3.25 How to exclude classes from component scan . . . 19
3.26 How does component scan work? . . . 19
3.27 What is the most challenging task you've done? . . . 19
3.28 What are the top 3 performance bottlenecks in microservices? . . . 19
3.29 How do you monitor microservices? . . . 20
3.30 How do you ensure system resiliency under high load? . . . 20
3.31 What is centralized configuration and secrets management? . . . 20
3.32 What is service discovery (Eureka/Consul)? . . . 20
3.33 Inter-service communication: Feign vs RestTemplate vs WebClient . . . 20
3.34 Circuit Breaker and Retry: Resilience4j . . . 20
3.35 Blue-green vs canary deployments . . . 21
3.36 Handling version mismatch between services . . . 21
3.37 What are Maven commands you use daily? . . . 21
3.38 Have you used Docker? Benefits and challenges? . . . 21

4 Spring & Bean Lifecycle **21**
4.1 Bean lifecycle and Spring container . . . 21
4.2 Use of @Component, @Service, @Repository . . . 21
4.3 Role of @ComponentScan, @Configuration, @Bean . . . 22
4.4 What is Spring Boot auto-configuration? . . . 22

5 JPA & Database **22**

5.1 What is Fetch Type (Lazy vs Eager Loading)? . . . 22
5.2 Lazy vs Eager loading — real-time use cases . . . 22
5.3 Complex entity relationship experience . . . 22
5.4 What is the N+1 query problem? . . . 22
5.5 How to optimize N+1 using Spring Data JPA . . . 23
5.6 Difference between get() and load() in Hibernate . . . 23
5.7 Writing optimized JPQL and Criteria queries . . . 23
5.8 Transaction management: @Transactional deep dive . . . 23

6 Testing & Mocking **23**
6.1 Differences between stubbing and mocking . . . 23
6.2 Why do we need both approaches? . . . 24
6.3 What is Spy in Mockito and when to use it? . . . 24
6.4 How to write JUnit test cases for static methods . . . 24
6.5 What is the use of Mockito framework? . . . 24

7 Coding & Problem Solving **24**
7.1 Reverse a string by preserving word position . . . 24
7.2 Remove duplicates from string/array/list . . . 25
7.3 Longest substring without repeating characters . . . 25
7.4 Check if two strings/numbers are palindrome . . . 25
7.5 Check if two strings/numbers are anagram . . . 26
7.6 Sort an array/list/string . . . 26
7.7 Count occurrence of characters in a string . . . 26
7.8 Print duplicate characters in a string . . . 27
7.9 Print only special characters in a string . . . 27
7.10 Reverse the given string . . . 27
7.11 Print difference between two strings . . . 27
7.12 Count words in a string . . . 28
7.13 Find second highest element in an array . . . 28
7.14 Find common elements of two arrays . . . 28
7.15 Decode string like a2b3c1 to aabbbc . . . 29
7.16 Check if a number is prime . . . 29
7.17 Generate Fibonacci series . . . 29
7.18 Find factorial of a number . . . 30
7.19 Find min and max element in an array . . . 30
7.20 Find max repeated word in a sentence . . . 30
7.21 Rotate array from k=2 . . . 30
7.22 Merge characters from two strings alternately . . . 31
7.23 Find max repeated word in a sentence . . . 31
7.24 Modify and improve given code . . . 31
7.25 Find a file in a subdirectory . . . 32
7.26 Write a method to fetch employee details . . . 32

8 Angular & Frontend **32**
8.1 What is ng-content and how does content projection work? . . . 32
8.2 Difference between @ViewChild and @ContentChild . . . 32
8.3 Explain Dependency Injection in Angular . . . 33
8.4 How does Angular load dynamic components? . . . 33
8.5 What is the PipeTransform interface? . . . 33
8.6 How does Angular bootstrapping work via AppModule? . . . 34
8.7 What is an HTTP interceptor? . . . 34

8.8 Common use cases of HTTP interceptors 34
8.9 What is GraphQL and how does it compare to REST? 34
8.10 Use Apollo in Angular to fetch GraphQL data 34
8.11 Differences: BehaviorSubject, Subject, ReplaySubject 35
8.12 Explain: switchMap, mergeMap, concatMap, exhaustMap 35
8.13 Design a semantic HTML navigation menu 35
8.14 List 5 semantic HTML tags and their uses 35
8.15 Fetch API data and display in a table 35
8.16 Search by name using Reactive Form 36
8.17 Error handling on API failure 36
8.18 Flatten a nested array 36

9 Miscellaneous **36**
9.1 What do you know about ISO8583? 36
9.2 How do Angular applications interact with backend APIs? 36
9.3 How do you clone code from Git and commit changes? 37
9.4 How to resolve merge conflicts? 37
9.5 Daily Git commands you use 37

10 Additional Frequently Asked Questions **37**
10.1 What is the difference between checked and unchecked exceptions? 37
10.2 What is the purpose of @Autowired annotation in Spring? 37
10.3 What is the Circuit Breaker pattern in microservices? 37

Introduction

This document provides answers and examples for Java, Spring Boot, Microservices, and Angular interview questions, organized by topic. Each question includes a beginner-friendly explanation and a practical example, suitable for interview preparation. Additional common questions are included at the end.

1 Core Java & OOPs

1.1 Difference between String, StringBuilder, and StringBuffer

Answer: String is immutable, creating new objects for modifications. StringBuilder is mutable, non-thread-safe, and fast. StringBuffer is mutable, thread-safe, but slower due to synchronization.

```java
public class StringExample {
  public static void main(String[] args) {
    String str = "Hello";
    str += " World"; // New object
    System.out.println(str); // Hello World
    StringBuilder sb = new StringBuilder("Hello");
    sb.append(" World");
    System.out.println(sb); // Hello World
    StringBuffer sbf = new StringBuffer("Hello");
    sbf.append(" World");
    System.out.println(sbf); // Hello World
  }
}
```

1.2 Difference between .equals() method and == operator

Answer: == compares object references or primitive values. .equals() compares object content, customizable for classes.

```java
public class EqualsExample {
  public static void main(String[] args) {
    String s1 = new String("Hello");
    String s2 = new String("Hello");
    System.out.println(s1 == s2); // false
    System.out.println(s1.equals(s2)); // true
    String s3 = "Hello";
    String s4 = "Hello";
    System.out.println(s3 == s4); // true
  }
}
```

1.3 Difference between method overloading and method overriding

Answer: Overloading uses different parameters in the same class (compile-time). Overriding redefines a method in a subclass (runtime).

```java
class Animal {
  void sound() { System.out.println("Animal sound"); }
}
class Dog extends Animal {
  void sound() { System.out.println("Dog barks"); } // Override
  void sound(String type) { System.out.println("Dog " + type); } //
     Overload
}
public class PolymorphismExample {
  public static void main(String[] args) {
    Dog dog = new Dog();
    dog.sound(); // Dog barks
    dog.sound("growls"); // Dog growls
  }
}
```

1.4 What is Singleton class?

Answer: Ensures one instance with global access, used for shared resources.

1.5 Create Singleton class

Answer: Uses private constructor, static instance, and static access method.

```java
public class Singleton {
  private static Singleton instance;
  private Singleton() {}
  public static synchronized Singleton getInstance() {
    if (instance == null) instance = new Singleton();
    return instance;
  }
  public static void main(String[] args) {
    Singleton s1 = Singleton.getInstance();
    Singleton s2 = Singleton.getInstance();
    System.out.println(s1 == s2); // true
  }
```

```
}
```

1.6 What is thread safety and how do you ensure thread-safe classes?

Answer: Thread safety prevents data corruption in multi-threaded environments using synchronization, immutability, or thread-safe classes.

```
public class ThreadSafeCounter {
  private int count = 0;
  public synchronized void increment() { count++; }
  public synchronized int getCount() { return count; }
  public static void main(String[] args) throws InterruptedException {
    ThreadSafeCounter counter = new ThreadSafeCounter();
    Runnable task = () -> { for (int i = 0; i < 1000; i++) counter.
        increment(); };
    Thread t1 = new Thread(task); Thread t2 = new Thread(task);
    t1.start(); t2.start(); t1.join(); t2.join();
    System.out.println(counter.getCount()); // 2000
  }
}
```

1.7 How does HashMap work internally?

Answer: Uses a hash table with buckets. Keys' hashCode() determines bucket index; collisions use linked lists or trees (Java 8+).

```
import java.util.HashMap;
public class HashMapExample {
  public static void main(String[] args) {
    HashMap<String, Integer> map = new HashMap<>();
    map.put("A", 1);
    map.put("B", 2);
    System.out.println(map.get("A")); // 1
  }
}
```

1.8 Difference: HashMap vs Hashtable vs ConcurrentHashMap

Answer: HashMap: non-thread-safe, allows nulls. Hashtable: thread-safe, no nulls. ConcurrentHashMap: thread-safe, concurrent access, no nulls.

```
import java.util.*;
public class MapComparison {
  public static void main(String[] args) {
    HashMap<String, Integer> hm = new HashMap<>();
    hm.put(null, 1);
    System.out.println(hm); // {null=1}
    Hashtable<String, Integer> ht = new Hashtable<>();
    ConcurrentHashMap<String, Integer> chm = new ConcurrentHashMap<>();
  }
}
```

1.9 How does HashMap work with Employee object as key?

Answer: Uses Employee's hashCode() and equals() for bucket placement and collision resolution.

```java
import java.util.HashMap;
class Employee {
  int id; String name;
  Employee(int id, String name) { this.id = id; this.name = name; }
  @Override public int hashCode() { return id * 31 + name.hashCode(); }
  @Override public boolean equals(Object obj) {
    if (!(obj instanceof Employee)) return false;
    Employee other = (Employee) obj;
    return id == other.id && name.equals(other.name);
  }
}
public class HashMapEmployee {
  public static void main(String[] args) {
    HashMap<Employee, String> map = new HashMap<>();
    Employee e1 = new Employee(1, "Alice");
    map.put(e1, "Developer");
    System.out.println(map.get(new Employee(1, "Alice"))); // Developer
  }
}
```

1.10 What is immutability and how does it help in concurrency?

Answer: Immutability prevents state changes, ensuring thread safety without synchronization.

```java
public final class ImmutableClass {
  private final int value;
  public ImmutableClass(int value) { this.value = value; }
  public int getValue() { return value; }
  public static void main(String[] args) {
    ImmutableClass obj = new ImmutableClass(42);
    System.out.println(obj.getValue()); // 42
  }
}
```

1.11 What is volatile and synchronized?

Answer: volatile ensures variable visibility; synchronized ensures mutual exclusion.

```java
public class VolatileSynchronized {
  private volatile boolean running = true;
  public synchronized void update() { running = false; }
  public static void main(String[] args) throws InterruptedException {
    VolatileSynchronized vs = new VolatileSynchronized();
    new Thread(() -> { while (vs.running) {} System.out.println("Stopped");
        }).start();
    Thread.sleep(1000);
    vs.update();
  }
}
```

1.12 How many ways can an object be created in Java?

Answer: Using new, Class.forName(), clone(), deserialization, factory methods.

```java
public class ObjectCreation {
  public static void main(String[] args) throws Exception {
    ObjectCreation obj1 = new ObjectCreation();
    ObjectCreation obj2 = (ObjectCreation) Class.forName("ObjectCreation").
        newInstance();
  }
}
```

1.13 What is the use of ResponseEntity?

Answer: Represents HTTP response with status, headers, and body.

```java
import org.springframework.http.*;
import org.springframework.web.bind.annotation.*;
@RestController
public class ResponseEntityExample {
  @GetMapping("/user")
  public ResponseEntity<String> getUser() {
    return new ResponseEntity<>("User found", HttpStatus.OK);
  }
}
```

1.14 What is meant by functional interfaces?

Answer: Interfaces with one abstract method, used with lambdas.

```java
@FunctionalInterface
interface MyFunction { void apply(String s); }
public class FunctionalInterfaceExample {
  public static void main(String[] args) {
    MyFunction func = s -> System.out.println(s);
    func.apply("Hello"); // Hello
  }
}
```

1.15 How do functional interfaces work?

Answer: Enable functional programming via lambda expressions.

```java
import java.util.function.Consumer;
public class FunctionalInterfaceWork {
  public static void main(String[] args) {
    Consumer<String> consumer = s -> System.out.println(s);
    consumer.accept("Hello Consumer"); // Hello Consumer
  }
}
```

1.16 What is dependency injection and its types?

Answer: Provides dependencies externally. Types: constructor, setter, field injection.

```java
import org.springframework.stereotype.Component;
import org.springframework.beans.factory.annotation.Autowired;
@Component
class Service { public void serve() { System.out.println("Serving"); } }
@Component
```

```
class Client {
  private final Service service;
  @Autowired
  public Client(Service service) { this.service = service; }
  public void doWork() { service.serve(); }
}
```

1.17 Difference between IOC and Dependency Injection

Answer: IoC inverts control to a framework; DI is a way to achieve IoC.

```
// See above (Question 16)
```

1.18 How many ways to achieve dependency injection and which is best?

Answer: Constructor, setter, field injection. Constructor is best for explicit dependencies.

```
// See Question 16
```

1.19 @Primary vs @Qualifier — which takes priority?

Answer: @Primary sets default bean; @Qualifier overrides it.

```
import org.springframework.stereotype.Component;
import org.springframework.beans.factory.annotation.*;
interface Service {}
@Component @Primary
class DefaultService implements Service {}
@Component @Qualifier("special")
class SpecialService implements Service {}
@Component
class Client {
  @Autowired @Qualifier("special") Service service;
}
```

1.20 How to create beans manually?

Answer: Use @Bean in a @Configuration class.

```
import org.springframework.context.annotation.*;
@Configuration
public class AppConfig {
  @Bean
  public Service myService() { return new Service(); }
}
```

1.21 What is the difference between @RestController and @Controller?

Answer: @Controller returns view names; @RestController returns data (e.g., JSON).

```
import org.springframework.stereotype.*;
import org.springframework.web.bind.annotation.*;
@Controller
public class MyController {
  @GetMapping("/view")
```

```
  public String getView() { return "view"; }
}
@RestController
public class MyRestController {
  @GetMapping("/api")
  public String getData() { return "Data"; }
}
```

1.22 What is @SpringBootApplication annotation?

Answer: Combines @EnableAutoConfiguration, @ComponentScan, @Configuration.

```
import org.springframework.boot.*;
import org.springframework.boot.autoconfigure.*;
@SpringBootApplication
public class Application {
  public static void main(String[] args) {
    SpringApplication.run(Application.class, args);
  }
}
```

1.23 What are stereotype annotations in Spring Boot?

Answer: @Component, @Service, @Repository, @Controller mark classes for scanning.

```
import org.springframework.stereotype.*;
@Service
public class MyService {
  public void serve() { System.out.println("Service"); }
}
```

1.24 What is path variable?

Answer: @PathVariable extracts URL path values.

```
import org.springframework.web.bind.annotation.*;
@RestController
public class PathVariableExample {
  @GetMapping("/user/{id}")
  public String getUser(@PathVariable int id) {
    return "User ID: " + id;
  }
}
```

1.25 What is the default server in Spring Boot?

Answer: Tomcat.

```
// Run Application.java to start Tomcat
```

1.26 What is the default port in Spring Boot?

Answer: 8080.

```
// Access http://localhost:8080
```

2 Java 8 Features

2.1 Java 8 features used in your project

Answer: Lambda expressions, Stream API, Optional, default/static methods, forEach.

```
import java.util.*;
public class Java8Features {
  public static void main(String[] args) {
    List<String> list = Arrays.asList("A", "B");
    list.forEach(s -> System.out.println(s)); // A, B
  }
}
```

2.2 What is a lambda expression?

Answer: Concise function representation: (params) -> expression.

```
import java.util.*;
public class LambdaExample {
  public static void main(String[] args) {
    List<String> list = Arrays.asList("A", "B");
    list.forEach(s -> System.out.println(s)); // A, B
  }
}
```

2.3 Write code using lambda expression

```
import java.util.*;
public class LambdaCode {
  public static void main(String[] args) {
    List<Integer> numbers = Arrays.asList(1, 2, 3);
    numbers.forEach(n -> System.out.println(n * 2)); // 2, 4, 6
  }
}
```

2.4 What is Stream API?

Answer: Processes collections functionally with operations like filter, map, collect.

```
import java.util.*;
import java.util.stream.*;
public class StreamExample {
  public static void main(String[] args) {
    List<Integer> numbers = Arrays.asList(1, 2, 3, 4);
    List<Integer> evens = numbers.stream()
      .filter(n -> n % 2 == 0)
      .collect(Collectors.toList());
    System.out.println(evens); // [2, 4]
  }
}
```

2.5 Intermediate vs terminal operations in streams

Answer: Intermediate (lazy, e.g., filter, map); terminal (triggers, e.g., collect, forEach).

```
import java.util.*;
public class StreamOperations {
  public static void main(String[] args) {
    List<Integer> numbers = Arrays.asList(1, 2, 3);
    numbers.stream().filter(n -> n > 1).forEach(System.out::println); // 2,
        3
  }
}
```

2.6 List of intermediate and terminal methods

Answer: Intermediate: filter, map, sorted, distinct. Terminal: collect, forEach, reduce, count.

```
import java.util.*;
import java.util.stream.*;
public class StreamMethods {
  public static void main(String[] args) {
    List<Integer> numbers = Arrays.asList(1, 2, 2, 3);
    long count = numbers.stream().distinct().count(); // 3
    System.out.println(count);
  }
}
```

2.7 Why Java introduced default and static methods in interfaces

Answer: Default: Add methods without breaking implementations. Static: Utility methods.

```
interface MyInterface {
  default void defaultMethod() { System.out.println("Default"); }
  static void staticMethod() { System.out.println("Static"); }
}
class MyClass implements MyInterface {}
public class InterfaceExample {
  public static void main(String[] args) {
    new MyClass().defaultMethod(); // Default
    MyInterface.staticMethod(); // Static
  }
}
```

2.8 How to create an immutable class in Java

Answer: Use final class, final fields, no setters, deep copy for mutable objects.

```
public final class ImmutableClass {
  private final int value;
  public ImmutableClass(int value) { this.value = value; }
  public int getValue() { return value; }
  public static void main(String[] args) {
    ImmutableClass obj = new ImmutableClass(42);
    System.out.println(obj.getValue()); // 42
  }
}
```

2.9 How to use groupingBy() in streams

Answer: Groups elements by a classifier.

```java
import java.util.*;
import java.util.stream.*;
public class GroupingByExample {
  public static void main(String[] args) {
    List<String> names = Arrays.asList("Alice", "Bob", "Adam");
    Map<Character, List<String>> grouped = names.stream()
      .collect(Collectors.groupingBy(s -> s.charAt(0)));
    System.out.println(grouped); // {A=[Alice, Adam], B=[Bob]}
  }
}
```

2.10 How to create an Optional of an employee object

Answer: Wraps object to handle null cases.

```java
import java.util.Optional;
class Employee {
  String name;
  Employee(String name) { this.name = name; }
}
public class OptionalExample {
  public static void main(String[] args) {
    Optional<Employee> emp = Optional.of(new Employee("Alice"));
    System.out.println(emp.get().name); // Alice
  }
}
```

2.11 Use Java 8 streams to remove duplicates from a list

```java
import java.util.*;
import java.util.stream.*;
public class RemoveDuplicates {
  public static void main(String[] args) {
    List<Integer> numbers = Arrays.asList(1, 2, 2, 3);
    List<Integer> unique = numbers.stream().distinct().collect(Collectors.
       toList());
    System.out.println(unique); // [1, 2, 3]
  }
}
```

2.12 Find 3rd highest salary using Java 8 streams

```java
import java.util.*;
import java.util.stream.*;
class Employee {
  double salary;
  Employee(double salary) { this.salary = salary; }
  double getSalary() { return salary; }
}
public class ThirdHighestSalary {
  public static void main(String[] args) {
    List<Employee> employees = Arrays.asList(
```

```
      new Employee(50000), new Employee(70000), new Employee(60000)
    );
    double thirdHighest = employees.stream()
      .map(Employee::getSalary)
      .distinct()
      .sorted(Comparator.reverseOrder())
      .skip(2)
      .findFirst()
      .orElse(0.0);
    System.out.println(thirdHighest); // 0.0 (if <3 salaries)
  }
}
```

3 Spring Boot & Microservices

3.1 How do you configure different environments in Spring Boot?

Answer: Use application-{profile}.properties and spring.profiles.active.

```
# application-dev.properties
server.port=8081
```

```
import org.springframework.context.annotation.*;
@Profile("dev")
@Component
public class DevConfig {}
```

3.2 What is a Spring Boot profile and how did you use it?

Answer: Profiles activate environment-specific configurations.

```
# application.properties
spring.profiles.active=dev
```

3.3 What is exception handling and what is an advisor?

Answer: Exception handling manages errors; advisor combines AOP advice and pointcut.

```
import org.springframework.http.*;
import org.springframework.web.bind.annotation.*;
@RestController
public class ExceptionController {
  @ExceptionHandler(NullPointerException.class)
  public ResponseEntity<String> handleNPE() {
    return new ResponseEntity<>("Error", HttpStatus.BAD_REQUEST);
  }
}
```

3.4 How does @Transactional annotation work?

Answer: Manages transactions, ensuring atomicity.

```java
import org.springframework.stereotype.*;
import org.springframework.transaction.annotation.*;
@Service
public class UserService {
  @Transactional
  public void saveUser() {}
}
```

3.5 What are the design patterns used in microservices?

Answer: Circuit Breaker, API Gateway, Service Discovery, Event-Driven, CQRS.

3.6 Explain each design pattern and when to use them

Answer: Circuit Breaker: Prevents failures. API Gateway: Routing/security. Service Discovery: Dynamic location. Event-Driven: Loose coupling. CQRS: Complex domains.

```java
import io.github.resilience4j.circuitbreaker.annotation.*;
@Service
public class MyService {
  @CircuitBreaker(name = "myService")
  public String callApi() { return "Success"; }
}
```

3.7 What is microservices architecture?

Answer: Small, independent services communicating via APIs.

```java
import org.springframework.web.bind.annotation.*;
@RestController
public class UserController {
  @GetMapping("/users")
  public String getUsers() { return "User List"; }
}
```

3.8 Why microservices?

Answer: Scalability, independent deployment, fault isolation.

3.9 How do microservices communicate with each other?

Answer: Via REST, message queues, or gRPC.

```java
import org.springframework.web.client.RestTemplate;
public class Communication {
  public static void main(String[] args) {
    RestTemplate restTemplate = new RestTemplate();
    String response = restTemplate.getForObject("http://other-service/users
        ", String.class);
  }
}
```

3.10 Difference between synchronous and asynchronous communication

Answer: Synchronous: Blocking (REST). Asynchronous: Non-blocking (Kafka).

3.11 When to use synchronous vs asynchronous communication

Answer: Synchronous for immediate responses; asynchronous for decoupled systems.

```
import org.springframework.kafka.annotation.*;
@Service
public class KafkaConsumer {
  @KafkaListener(topics = "myTopic")
  public void consume(String message) {
    System.out.println(message);
  }
}
```

3.12 What is Kafka and how have you implemented it?

Answer: Distributed messaging system for event-driven architectures.

```
import org.springframework.kafka.core.*;
import org.springframework.stereotype.*;
@Service
public class KafkaProducer {
  @Autowired
  private KafkaTemplate<String, String> kafkaTemplate;
  public void sendMessage(String msg) {
    kafkaTemplate.send("myTopic", msg);
  }
}
```

3.13 How have you used Spring Security in your project?

Answer: Secures applications via authentication/authorization.

```
import org.springframework.context.annotation.*;
import org.springframework.security.config.annotation.web.builders.*;
@Configuration
@EnableWebSecurity
public class SecurityConfig {
  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
      Exception {
    http.authorizeRequests().anyRequest().authenticated().and().httpBasic()
        ;
    return http.build();
  }
}
```

3.14 What is JWT security and how have you used it?

Answer: Token-based authentication using JSON Web Tokens.

```
import io.jsonwebtoken.*;
public class JwtExample {
  public String generateToken(String username) {
    return Jwts.builder()
      .setSubject(username)
      .signWith(SignatureAlgorithm.HS512, "secret")
      .compact();
  }
```

```
}
```

3.15 What are the things to consider while developing REST APIs?

Answer: RESTful principles, versioning, error handling, security, documentation.

```
import org.springframework.http.*;
import org.springframework.web.bind.annotation.*;
@RestController
@RequestMapping("/api/v1")
public class ApiController {
  @GetMapping("/users")
  public ResponseEntity<List<String>> getUsers() {
    return ResponseEntity.ok(Arrays.asList("Alice", "Bob"));
  }
}
```

3.16 Difference between PUT, POST, and PATCH

Answer: POST creates, PUT updates entire resource, PATCH updates partially.

```
import org.springframework.web.bind.annotation.*;
@RestController
public class UserController {
  @PostMapping("/users") public String createUser() { return "Created"; }
  @PutMapping("/users/{id}") public String updateUser(@PathVariable int id)
       { return "Updated"; }
  @PatchMapping("/users/{id}") public String patchUser(@PathVariable int id
      ) { return "Patched"; }
}
```

3.17 API versioning strategies

Answer: URI versioning, query parameters, headers, media types.

```
import org.springframework.web.bind.annotation.*;
@RestController
@RequestMapping("/api/v1")
public class VersionedController {
  @GetMapping("/users")
  public String getUsers() { return "Version 1"; }
}
```

3.18 What is idempotency and why is it critical?

Answer: Ensures multiple identical requests have the same effect, critical for reliability.

```
import org.springframework.web.bind.annotation.*;
@RestController
public class IdempotentController {
  @PutMapping("/users/{id}")
  public ResponseEntity<String> updateUser(@PathVariable int id) {
    return ResponseEntity.ok("Updated");
  }
}
```

3.19 How do you achieve idempotency in microservices?

Answer: Use unique request IDs or design idempotent operations.

```
import java.util.*;
@Service
public class IdempotentService {
  private Set<String> processedIds = new HashSet<>();
  public String process(String requestId) {
    if (processedIds.contains(requestId)) return "Already processed";
    processedIds.add(requestId);
    return "Processed";
  }
}
```

3.20 Securing APIs using JWT and OAuth2

Answer: JWT for authentication, OAuth2 for authorization.

```
// See JWT example (Question 14)
```

3.21 Validating incoming payloads with annotations

Answer: Use @Valid and Bean Validation annotations.

```
import javax.validation.constraints.*;
public class User {
  @NotNull private String name;
}
import org.springframework.web.bind.annotation.*;
import javax.validation.*;
@RestController
public class UserController {
  @PostMapping("/users")
  public ResponseEntity<String> createUser(@Valid @RequestBody User user) {
    return ResponseEntity.ok("Valid");
  }
}
```

3.22 What are the steps to test Spring Boot microservices applications?

Answer: Unit tests, integration tests, mock external services, test REST APIs.

```
import org.junit.jupiter.api.*;
import org.springframework.boot.test.context.*;
import org.springframework.boot.test.web.client.*;
@SpringBootTest
public class UserControllerTest {
  @Autowired private TestRestTemplate restTemplate;
  @Test
  public void testGetUsers() {
    ResponseEntity<String> response = restTemplate.getForEntity("/users",
        String.class);
    Assertions.assertEquals(HttpStatus.OK, response.getStatusCode());
  }
}
```

3.23 How to handle exceptions in Spring Boot

Answer: Use @ExceptionHandler or @ControllerAdvice.

```
import org.springframework.http.*;
import org.springframework.web.bind.annotation.*;
@ControllerAdvice
public class GlobalExceptionHandler {
  @ExceptionHandler(Exception.class)
  public ResponseEntity<String> handleException(Exception e) {
    return new ResponseEntity<>("Error: " + e.getMessage(), HttpStatus.
        INTERNAL_SERVER_ERROR);
  }
}
```

3.24 How to create global exceptions and what annotations are used

Answer: Use @ControllerAdvice and @ExceptionHandler.

```
// See above
```

3.25 How to exclude classes from component scan

Answer: Use exclude or excludeFilters in @ComponentScan.

```
import org.springframework.context.annotation.*;
@ComponentScan(basePackages = "com.example", excludeFilters = @Filter(type
    = FilterType.ASSIGNABLE_TYPE, classes = MyClass.class))
@Configuration
public class AppConfig {}
```

3.26 How does component scan work?

Answer: Scans packages for stereotype-annotated classes, registering them as beans.

```
import org.springframework.stereotype.*;
@Component
public class MyComponent {}
```

3.27 What is the most challenging task you've done?

Answer: Optimizing a microservice for high load with caching.

```
import org.springframework.cache.annotation.*;
@Service
public class OptimizedService {
  @Cacheable("data")
  public String getData() { return "Data"; }
}
```

3.28 What are the top 3 performance bottlenecks in microservices?

Answer: Network latency, database queries, resource management.

```java
import org.springframework.data.jpa.repository.*;
public interface UserRepository extends JpaRepository<User, Long> {
  @Query("SELECT u FROM User u WHERE u.active = true")
  List<User> findActiveUsers();
}
```

3.29 How do you monitor microservices?

Answer: Use Prometheus, Grafana, or Spring Actuator.

```
# application.properties
management.endpoints.web.exposure.include=*
```

3.30 How do you ensure system resiliency under high load?

Answer: Use circuit breakers, retries, load balancing, caching.

```java
// See Circuit Breaker (Question 6)
```

3.31 What is centralized configuration and secrets management?

Answer: Centralized configuration (Spring Cloud Config); secrets (Vault).

```
# application.properties
spring.config.import=configserver:http://config-server
```

3.32 What is service discovery (Eureka/Consul)?

Answer: Dynamically locates services.

```java
import org.springframework.cloud.netflix.eureka.*;
@SpringBootApplication
@EnableEurekaClient
public class Application {}
```

3.33 Inter-service communication: Feign vs RestTemplate vs WebClient

Answer: RestTemplate (synchronous), WebClient (reactive), Feign (declarative).

```java
import feign.*;
@FeignClient(name = "user-service")
interface UserClient {
  @GetMapping("/users")
  List<String> getUsers();
}
```

3.34 Circuit Breaker and Retry: Resilience4j

Answer: Circuit Breaker prevents failures; Retry attempts failed operations.

```java
import io.github.resilience4j.circuitbreaker.annotation.*;
import io.github.resilience4j.retry.annotation.*;
@Service
public class ResilientService {
  @CircuitBreaker(name = "myService")
```