💻 Core Java

What are the key differences between ArrayList and LinkedList?

ArrayList -> -It is slow to add element at rare position as compared to LinkedList.

-Array use dynamic Array data structure

-It cannot use discrete memory

-ArrayList doesn't support linke

-List Interface implement

LinkedList -> -It is faster for data operations as compare to arrylist

-LinkedList use doubly linked list data structure

-it uesd discrete memory

-it support link with multiple nodes.

-it implement list and queue

Explain the concept of immutability in Java. How do you create an immutable class?

**🔒 Why Use Immutable Objects?**

Thread safety: No synchronization needed.

Predictability: Easier to reason about code.

Security: Prevents unintended changes.

Caching and optimization: Safe to cache and reuse.

**🛠️ How to Create an Immutable Class in Java**

To make a class immutable, follow these steps:

Make the class final so it can't be subclassed.

Make all fields private and final.

Don't provide setters.

Initialize all fields via constructor.

Perform deep copies of mutable objects in constructor and getters.

Avoid exposing internal references to mutable objects.

Example ->

public final class Employee {

private final String name;

private final int id;

private final List<String> skills;

public Employee(String name, int id, List<String> skills) {

this.name = name;

this.id = id;

// Defensive copy to ensure immutability

this.skills = new ArrayList<>(skills);

}

public String getName() {

return name;

}

public int getId() {

return id;

}

public List<String> getSkills() {

// Return a copy to prevent external modification

return new ArrayList<>(skills);

}

}

What is the difference between == and .equals() in Java?

== check the indexing and .equals() check value

How does Java handle memory management and garbage collection?

🧠 Memory Management in Java

Java memory is divided into several regions:

**Heap Memory**

Stores objects and class instances.

Managed by the Garbage Collector (GC).

Divided into:

Young Generation (Eden + Survivor spaces)

Old Generation (Tenured space)

**Stack Memory**

Stores method calls and local variables.

Each thread has its own stack.

Memory is automatically freed when a method call ends.

**Method Area (MetaSpace in Java 8+)**

Stores class metadata, static variables, and method definitions.

**Program Counter (PC) Register**

Keeps track of the current instruction being executed.

**Native Method Stack**

Used for native (non-Java) method calls.

🧹 **Garbage Collection (GC)**

Garbage Collection is the process of automatically identifying and removing unused objects from memory to free up space.

🔄 GC Phases:

Mark – Identify which objects are still in use.

Sweep – Remove unreferenced objects.

Compact – Rearrange memory to avoid fragmentation (optional, depends on GC algorithm).

Java 8 supports several GC algorithms:

-------------------------------------------------------------------------------------------

GC Type Description |

Serial GC | Single-threaded, best for small applications.

Parallel GC | Multi-threaded, good for throughput.

CMS (Concurrent Mark Sweep) | Low pause times, deprecated in later versions. and removed in java 14

G1 GC (Garbage First) | Default in Java 8+, balances pause time and throughput.

What are functional interfaces and how are they used in Java 8?

A functional interface is an interface that contains exactly one abstract method. It may have multiple default or static methods, but only one method must be abstract.

✅ Example:

@FunctionalInterface

interface MyFunction {

void execute();

}

🔍 What Is the Function<T, R> Interface?

java

@FunctionalInterface

public interface Function<T, R> {

R apply(T t);

}

T: Input type

R: Return type

The apply() method is the single abstract method that must be implemented.

🧪 Example Usage

✅ Basic Transformation

Function<String, Integer> stringLength = str -> str.length();

System.out.println(stringLength.apply("Java")); // Output: 4

✅ Chaining Functions

You can chain multiple functions using andThen() and compose():

java

Function<Integer, Integer> multiply = x -> x \* 2;

Function<Integer, Integer> add = x -> x + 3;

Function<Integer, Integer> combined = multiply.andThen(add);

System.out.println(combined.apply(5)); // Output: 13

🧠 Real-World Use Case

✅ Map.computeIfAbsent() with Function

Map<String, Integer> nameMap = new HashMap<>();

Integer value = nameMap.computeIfAbsent("John", s -> s.length());

System.out.println(value); // Output: 4

Here, the Function<String, Integer> is used to compute a value if the key is missing.

🧩 Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a way of writing programs where you build your code using objects. An object is like a mini-program that has:

Data (called fields or variables)

Actions (called methods or functions)

Think of it like a real-world object: a car has data (color, speed) and actions (drive, stop). In OOP, you create software using these kinds of objects.

Can you explain the four pillars of OOP with examples?

What is polymorphism and how is it implemented in Java?

**Polymorphism** means “many forms.” In programming, it allows one thing—like a method or object—to behave differently based on the context.

Imagine you have a remote control. You press the power button:

On a TV, it turns on the screen.

On an AC, it starts cooling.

On a fan, it starts spinning.

Same button, different actions. That’s polymorphism!

How do you achieve encapsulation in your code?

**Encapsulation** means:

Wrapping data (fields) and methods (functions) that operate on that data into a single unit (a class).

Restricting direct access to some parts of an object to keep it safe from unintended changes.

How to Achieve Encapsulation in Java

### ✅ 1. Make fields private

This hides the data from outside access.

### ✅ 2. Provide public getter and setter methods

These methods allow controlled access to the data.

### 🧪 Example:

public class Person {

private String name; // private field

private int age;

// Public getter

public String getName() {

return name;

}

// Public setter

public void setName(String name) {

if (name != null && !name.isEmpty()) {

this.name = name;

}

}

public int getAge() {

return age;

}

public void setAge(int age) {

if (age >= 0) {

this.age = age;

}

}}

### ✅ Usage:

Person p = new Person();p.setName("Alice");p.setAge(25);

System.out.println(p.getName()); // Output: Alice

What is the difference between abstract classes and interfaces?

| **Feature** | **Abstract Class** | **Interface** |
| --- | --- | --- |
| **Purpose** | Partial abstraction | Full abstraction |
| **Methods** | Can have both abstract and concrete methods | Only abstract methods (until Java 8+) |
| **Fields** | Can have instance variables | Only static and final constants |
| **Access Modifiers** | Can use any access modifier | Methods are public by default |
| **Inheritance** | A class can extend **only one** abstract class | A class can implement **multiple** interfaces |
| **Constructors** | Can have constructors | Cannot have constructors |
| **Default Methods (Java 8+)** | Not applicable | Can have default and static methods |

What is Spring Boot and how is it different from the traditional Spring framework?

**Spring Boot** is a framework built on top of the traditional Spring Framework. It simplifies the process of building production-ready applications by:

* **Auto-configuring** components based on the dependencies in your project
* Providing **starter templates** for common use cases (e.g., web, data, security)
* Embedding servers like **Tomcat** or **Jetty** so you don’t need to deploy WAR files

## Flexibility to configure applications using ****XML****, ****annotations****, or ****Java-based configuration**** 🔍 Key Differences

| **Feature** | **Spring Framework** | **Spring Boot** |
| --- | --- | --- |
| **Configuration** | Manual (XML, annotations, Java config) | Auto-configured based on dependencies |
| **Startup Time** | Slower due to setup | Faster with embedded server and defaults |
| **Deployment** | Requires external server (e.g., Tomcat) | Embedded server; runs as standalone JAR |
| **Complexity** | More boilerplate and setup | Minimal setup; opinionated defaults |
| **Use Case** | Large, complex enterprise apps | Microservices, REST APIs, rapid prototyping |

### How does dependency injection work in Spring? Dependency Injection (DI) in ****Spring**** is a powerful design pattern that allows the framework to ****automatically provide objects (dependencies)**** that a class needs, rather than the class creating them itself. This promotes ****loose coupling****, ****testability****, and ****clean architecture****. 1. ****Constructor Injection****

Dependencies are passed via the constructor:

@Componentpublic class UserService {

private final UserRepository userRepository;

@Autowired

public UserService(UserRepository userRepository) {

this.userRepository = userRepository;

}}

### 2. ****Setter Injection****

Dependencies are set via setter methods:

@Componentpublic class UserService {

private UserRepository userRepository;

@Autowired

public void setUserRepository(UserRepository userRepository) {

this.userRepository = userRepository;

}}

What are REST controllers and how do you expose APIs using Spring Boot?

A class annotated with @RestController tells Spring Boot:

* This class will handle **web requests**

The methods will return **data** (like JSON), not HTML views  
  
  
@RestController@RequestMapping("/api")public class UserController {

@GetMapping("/users")

public List<User> getAllUsers() {

return userService.getUsers();

}

@PostMapping("/users")

public String addUser(@RequestBody User user) {

userService.saveUser(user);

return "User added successfully";

}}

## 🔧 How to Expose APIs Using Spring Boot

### 1. ****Create a Spring Boot Project****

Use Spring Initializr and include the **Spring Web** dependency.

### 2. ****Define a Model Class****

public class User {

private int id;

private String name;

// getters and setters}

### 3. ****Create a REST Controller****

Use @RestController and map endpoints with annotations like:

* @GetMapping
* @PostMapping
* @PutMapping
* @DeleteMapping

### 4. ****Run the Application****

Spring Boot runs on an embedded server (like Tomcat), so you can test your API at:

<http://localhost:8080/api/users>  
 **Topic Vise Learning**

# 1. Exception Handling in Spring Boot

Spring Boot lets you centralize error handling using @ControllerAdvice and @ExceptionHandler. You define custom exception classes, then catch and transform them into meaningful HTTP responses.

java

// Define a custom exceptionpublic class ResourceNotFoundException extends RuntimeException {

public ResourceNotFoundException(String message) {

super(message);

}}

// Central error handler@RestControllerAdvicepublic class GlobalExceptionHandler {

@ExceptionHandler(ResourceNotFoundException.class)

public ResponseEntity<Map<String, String>> handleNotFound(ResourceNotFoundException ex) {

Map<String,String> error = Map.of(

"error", ex.getMessage(),

"timestamp", Instant.now().toString()

);

return ResponseEntity.status(HttpStatus.NOT\_FOUND).body(error);

}

@ExceptionHandler(Exception.class)

public ResponseEntity<Map<String, String>> handleGeneric(Exception ex) {

Map<String,String> error = Map.of(

"error", "Internal server error",

"timestamp", Instant.now().toString()

);

return ResponseEntity.status(HttpStatus.INTERNAL\_SERVER\_ERROR).body(error);

}}

# 2. @Autowired, @Component, and @Service

@Component marks any class as a Spring bean. @Service is a specialization of @Component for service-layer classes. @Autowired injects a bean into a field, setter, or constructor.

java

@Componentpublic class EmailClient {

public void send(String to, String content) { /\* ... \*/ }}

@Servicepublic class NotificationService {

private final EmailClient emailClient;

@Autowired

public NotificationService(EmailClient emailClient) {

this.emailClient = emailClient;

}

public void notify(String user, String message) {

emailClient.send(user, message);

}}

# 3. JPA vs Hibernate

| **Aspect** | **JPA** | **Hibernate** |
| --- | --- | --- |
| Nature | Specification (API) | One of the most popular implementations |
| Packaging | javax.persistence | org.hibernate |
| Query Language | JPQL (standard) | HQL + native SQL + extra features |
| Vendor Lock-In | Low (can switch providers) | Higher (Hibernate-specific extensions) |

# 4. Defining Entity Relationships in JPA

Use annotations on model classes to map relationships.

java

@Entitypublic class Department {

@Id @GeneratedValue

private Long id;

@OneToMany(mappedBy = "department", cascade = CascadeType.ALL)

private List<Employee> employees = new ArrayList<>();}

@Entitypublic class Employee {

@Id @GeneratedValue

private Long id;

@ManyToOne

@JoinColumn(name = "department\_id")

private Department department;}

# 5. Transactions in Spring

Annotate service methods with @Transactional to ensure atomic operations.

java

@Servicepublic class OrderService {

@Autowired

private OrderRepository orderRepo;

@Transactional

public void placeOrder(Order order) {

orderRepo.save(order);

// any other database operations

// If an exception is thrown, all operations roll back

}}

# 6. Custom Queries with Spring Data JPA

Use the @Query annotation to define JPQL or native SQL.

java

public interface UserRepository extends JpaRepository<User, Long> {

@Query("SELECT u FROM User u WHERE u.email = :email")

Optional<User> findByEmailJPQL(@Param("email") String email);

@Query(

value = "SELECT \* FROM users WHERE created\_at > :since",

nativeQuery = true

)

List<User> findRecentUsers(@Param("since") LocalDate since);}

# 7. Native Queries vs JPQL

| **Query Type** | **Pros** | **Cons** |
| --- | --- | --- |
| JPQL | Portable, entity-centric, compile-time validation | Limited to JPA syntax |
| Native SQL | Full SQL power, vendor-specific optimizations | Database-dependent, risk of SQL injection |

# 8. Key Principles of REST Architecture

Stateless interactions

Uniform interface (resources identified by URIs)

Cacheable responses

Client-server separation

Layered system

(Optionally) Code on demand

# 9. Versioning in REST APIs

Common strategies to evolve APIs without breaking clients:

URI versioning: GET /api/v1/users

Query parameter: GET /users?version=1

Custom header: Accept-Version: 2

Media type versioning: Accept: application/vnd.company.v2+json

# 10. HTTP Status Codes

Use standard codes to signal result:

| **Code** | **Meaning** |
| --- | --- |
| 200 | OK (successful GET/PUT) |
| 201 | Created (POST) |
| 204 | No Content (successful delete) |
| 400 | Bad Request |
| 401 | Unauthorized |
| 403 | Forbidden |
| 404 | Not Found |
| 409 | Conflict |
| 500 | Internal Server Error |

@GetMapping("/items/{id}")public ResponseEntity<Item> getItem(@PathVariable Long id) {

return itemService.findById(id)

.map(ResponseEntity::ok)

.orElse(ResponseEntity.status(HttpStatus.NOT\_FOUND).build());}

# 11. Authentication & Authorization in REST APIs

Use Spring Security with JWT or OAuth2:

@EnableWebSecuritypublic class SecurityConfig extends WebSecurityConfigurerAdapter {

@Override

protected void configure(HttpSecurity http) throws Exception {

http

.csrf().disable()

.authorizeRequests()

.antMatchers("/auth/\*\*").permitAll()

.anyRequest().authenticated()

.and()

.oauth2ResourceServer().jwt();

}}

Issue JWTs at /auth/login and protect endpoints with role-based access.

# 12. API Testing Tools

Postman for manual endpoints exploration & automation

Swagger UI / OpenAPI for live documentation & testing

REST Assured for Java-based integration tests

SoapUI for SOAP and REST functional tests

# 13. Designing a URL Shortening Service

Core steps:

Generate unique short keys (Base62, hash + counter).

Store mapping in a database (id, original\_url, key).

Redirect short key: SELECT original\_url FROM urls WHERE key = ?.

Scale via caching (Redis) and sharding.

@RestControllerpublic class UrlController {

@Autowired

UrlService svc;

@PostMapping("/shorten")

public ShortenDto shorten(@RequestBody UrlDto dto) {

return svc.shorten(dto.getUrl());

}

@GetMapping("/{key}")

public ResponseEntity<Void> redirect(@PathVariable String key) {

return svc.findOriginal(key)

.map(orig -> ResponseEntity.status(HttpStatus.FOUND)

.location(URI.create(orig)).build())

.orElse(ResponseEntity.notFound().build());

}}

# 14. Microservices vs Monolithic Architecture

Monolith: Single codebase, single deployment. Microservices: Multiple small services, independently deployable.

| **Aspect** | **Monolith** | **Microservices** |
| --- | --- | --- |
| Deployment | One JAR/​WAR | Many small services |
| Scalability | Vertical scaling | Horizontal per service |
| Teams | Centralized | Decentralized per service |
| Complexity | Simple initially | More moving parts |

# 15. Ensuring Scalability & Performance

Use load balancers (Nginx, ELB)

Cache hot data (Redis, Memcached)

Asynchronous processing (RabbitMQ, Kafka)

Database optimization (indexes, sharding, read-replicas)

Profile and monitor (Prometheus, Grafana, New Relic)

# 16. Caching in Backend Applications

Use Spring Cache Abstraction:

@SpringBootApplication@EnableCachingpublic class App { /\* ... \*/ }

@Servicepublic class ProductService {

@Cacheable("products")

public Product getById(Long id) { /\* expensive DB call \*/ }}

Cache in Redis or in-memory to speed up repeated reads.

# 17. Handling Concurrency & Avoiding Race Conditions

Use optimistic locking with @Version or synchronize critical sections.

@Entitypublic class Account {

@Id @GeneratedValue private Long id;

private BigDecimal balance;

@Version private Long version;}

@Servicepublic class AccountService {

@Transactional

public void withdraw(Long accountId, BigDecimal amount) {

Account a = repo.findById(accountId).orElseThrow();

a.setBalance(a.getBalance().subtract(amount));

// version mismatch will throw OptimisticLockingFailureException

}}

# 18. Unit Testing Frameworks

JUnit 5 for writing test cases

Mockito for mocking dependencies

@ExtendWith(MockitoExtension.class)public class OrderServiceTest {

@Mock private OrderRepository repo;

@InjectMocks private OrderService svc;

@Test

void placeOrder\_savesOrder() {

Order o = new Order(1L, "Item");

when(repo.save(o)).thenReturn(o);

assertEquals("Item", svc.placeOrder(o).getItem());

}}

# 19. Mocking Dependencies in Tests

Use Mockito’s @Mock and when(...).thenReturn(...) to isolate the class under test from its collaborators.

# 20. CI/CD with Jenkins or GitHub Actions

**Jenkinsfile** (Groovy):

groovy

pipeline {

agent any

stages {

stage('Build') { steps { sh 'mvn clean package' } }

stage('Test') { steps { sh 'mvn test' } }

stage('Deploy'){ steps { sh './deploy.sh' } }

}}

**GitHub Actions** (.github/workflows/ci.yml):

yaml

name: CIon: [push]jobs:

build:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v4

- name: Set up JDK 17

uses: actions/setup-java@v3

with: { java-version: '17' }

- name: Build with Maven

run: mvn clean package

- name: Run tests

run: mvn test

# 21. Environment-Specific Configurations in Spring Boot

Create separate property files named by profile:

src/main/resources/

├─ application.properties # defaults

├─ application-dev.properties # dev overrides

├─ application-prod.properties # prod overrides

Activate a profile via:

-Dspring.profiles.active=dev

--spring.profiles.active=prod

SPRING\_PROFILES\_ACTIVE=test environment variable

Use @Profile("dev") on beans to load them conditionally:

@Configuration@Profile("dev")public class DevConfig { /\* dev-only beans \*/ }

💻 Core Java

1. What is the difference between == and .equals() in Java?

== compares object references (memory addresses).

.equals() compares object content (defined by the class).

Example:

java

String a = new String("hello");

String b = new String("hello");

System.out.println(a == b); // false

System.out.println(a.equals(b)); // true

2. What is a functional interface?

An interface with exactly one abstract method.

Used in lambda expressions.

Example:

java

@FunctionalInterface

interface MyFunc {

void execute();

}

🧩 OOP Concepts

3. What is polymorphism in Java?

Ability of an object to take many forms.

Achieved via method overriding and method overloading.

Example:

java

class Animal {

void sound() { System.out.println("Animal sound"); }

}

class Dog extends Animal {

void sound() { System.out.println("Dog barks"); }

}

4. Difference between abstract class and interface?

Feature Abstract Class Interface

Methods Can have both Only abstract (until Java 8)

Multiple Inheritance Not supported Supported

Fields Can have fields Only constants

🌐 Spring Boot

5. What is Spring Boot and why is it used?

Spring Boot simplifies Spring app development by:

Auto-configuring beans

Embedded servers (Tomcat)

No XML configuration

It helps build standalone, production-ready applications quickly.

6. What is dependency injection?

A design pattern where objects are provided their dependencies.

Spring uses annotations like @Autowired to inject beans.

🗄️ JPA / Hibernate

7. What is the difference between JPA and Hibernate?

JPA is a specification.

Hibernate is an implementation of JPA.

JPA provides annotations like @Entity, @Id, @OneToMany.

8. How do you define a one-to-many relationship in JPA?

java

@Entity

class Department {

@OneToMany(mappedBy = "department")

List<Employee> employees;

}

@Entity

class Employee {

@ManyToOne

Department department;

}

🔌 REST APIs

9. What is REST and what are its key principles?

REST stands for Representational State Transfer.

Principles:

Statelessness

Resource-based URLs

Use of HTTP methods (GET, POST, PUT, DELETE)

Standard status codes

10. How do you handle exceptions in Spring Boot REST APIs?

Use @ControllerAdvice and @ExceptionHandler:

java

@ControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(ResourceNotFoundException.class)

public ResponseEntity<String> handleNotFound(ResourceNotFoundException ex) {

return ResponseEntity.status(HttpStatus.NOT\_FOUND).body(ex.getMessage());

}

}

⚙️ System Design Basics

11. How would you design a URL shortening service?

Components:

API to accept long URL and return short code

Database to store mappings

Redirect logic for short URL

Use hashing or base62 encoding for short codes.

12. What is caching and where would you use it?

Caching stores frequently accessed data in memory.

Improves performance and reduces DB load.

Use cases: product catalog, user sessions, API responses.

🧪 Testing

13. What is unit testing and which frameworks do you use?

Unit testing checks individual components.

Use JUnit for writing tests and Mockito for mocking dependencies.

14. How do you mock a service in a test?

java

@Mock

UserService userService;

@InjectMocks

UserController userController;

Would you like me to format this into a printable guide or expand it with advanced questions for senior roles?

🔧 Scenario 1: API Performance Issue

❓ Your REST API is taking too long to respond. How would you troubleshoot and optimize it?

✅ Answer:

Step 1: Profile the API

Use tools like Spring Actuator, logs, or APM tools (e.g., New Relic) to identify bottlenecks.

Step 2: Check Database Queries

Look for N+1 query problems or missing indexes.

Optimize queries or use pagination.

Step 3: Add Caching

Use Redis or in-memory caching for frequently accessed data.

Step 4: Reduce Payload

Avoid sending unnecessary fields in the response.

Step 5: Asynchronous Processing

Use @Async or message queues for non-critical tasks.

🧩 Scenario 2: Handling Concurrent Updates

❓ Two users try to update the same record simultaneously. How do you prevent data inconsistency?

✅ Answer:

Use Optimistic Locking with JPA:

java

@Version

private int version;

This adds a version column to the entity. If two updates happen, the second one will fail if the version has changed.

Alternatively, use Pessimistic Locking if data integrity is critical:

java

@Lock(LockModeType.PESSIMISTIC\_WRITE)

🔐 Scenario 3: Securing an API

❓ You need to secure an endpoint so only authenticated users can access it. How would you do that in Spring Boot?

✅ Answer:

Use Spring Security:

Add dependency: spring-boot-starter-security

Configure security:

java

@Configuration

public class SecurityConfig extends WebSecurityConfigurerAdapter {

@Override

protected void configure(HttpSecurity http) throws Exception {

http

.authorizeRequests()

.antMatchers("/api/secure/\*\*").authenticated()

.and()

.httpBasic();

}

}

For token-based auth, use JWT (JSON Web Tokens).

🗄️ Scenario 4: Database Migration

❓ You need to add a new column to a live production database. How do you handle this safely?

✅ Answer:

Use a migration tool like Flyway or Liquibase.

Steps:

Create a migration script: V2\_\_add\_column.sql

Test in staging environment.

Deploy during low-traffic hours.

Ensure backward compatibility in code (e.g., handle nulls).

🔄 Scenario 5: Retry Logic for External API

❓ Your service calls an external API that occasionally fails. How do you implement retry logic?

✅ Answer:

Use Spring Retry:

java

@Retryable(

value = { RemoteServiceException.class },

maxAttempts = 3,

backoff = @Backoff(delay = 2000)

)

public String callExternalApi() {

// logic

}

Alternatively, use Resilience4j for more advanced features like circuit breakers and rate limiting.

📦 Scenario 6: File Upload Handling

❓ How would you implement a secure file upload endpoint in Spring Boot?

✅ Answer:

Use MultipartFile in controller:

java

@PostMapping("/upload")

public ResponseEntity<?> uploadFile(@RequestParam("file") MultipartFile file) {

// validate and store

}

Security considerations:

Validate file type and size.

Store files outside the web root.

Rename files to avoid collisions.

Scan for malware if needed.

Would you like me to add more scenarios related to microservices, CI/CD, or real-time data processing?

Can you provide more scenario-based questions?

Absolutely! Here are more scenario-based interview questions and answers tailored for a Backend Java Developer with 2 years of experience, covering areas like microservices, error handling, deployment, logging, and scalability.

🧵 Scenario 7: Microservices Communication

❓ You have two microservices: Service A needs to call Service B. How would you implement this communication?

✅ Answer:

Use REST API for synchronous communication:

Use RestTemplate or WebClient (preferred for reactive).

For resilience:

Add retry, timeout, and circuit breaker using Resilience4j.

For asynchronous communication:

Use message queues like Kafka or RabbitMQ.

Ensure authentication between services using JWT or API keys.

🧱 Scenario 8: Breaking a Monolith

❓ Your team wants to break a monolithic app into microservices. What steps would you take?

✅ Answer:

Identify bounded contexts: Split by business domains.

Extract services gradually: Start with least dependent modules.

Define APIs: Clear contracts between services.

Use shared libraries: For common utilities and models.

Implement service discovery: Use tools like Eureka or Consul.

Centralized logging and monitoring: Use ELK stack or Prometheus + Grafana.

🧼 Scenario 9: Cleaning Up Stale Data

❓ You need to delete stale records from the database every night. How would you implement this?

✅ Answer:

Use Spring Scheduler:

java

@Scheduled(cron = "0 0 2 \* \* ?") // Runs at 2 AM daily

public void cleanStaleData() {

// delete logic

}

Ensure:

Proper indexing for delete queries.

Logging and alerting on failures.

Transactional safety.

📊 Scenario 10: Logging Strategy

❓ How would you implement logging in a production-grade backend application?

✅ Answer:

Use SLF4J with Logback or Log4j2.

Log levels:

INFO for business flow

DEBUG for dev diagnostics

ERROR for failures

Use MDC (Mapped Diagnostic Context) for tracing request IDs.

Centralize logs using ELK stack or Fluentd + CloudWatch.

🧠 Scenario 11: Handling NullPointerException

❓ Your application throws a NullPointerException in production. How do you debug and prevent it?

✅ Answer:

Debugging:

Check logs and stack trace.

Identify the object that’s null.

Prevention:

Use Optional where applicable.

Add null checks or use Objects.requireNonNull().

Write unit tests to cover edge cases.

🚀 Scenario 12: Deployment Failure

❓ Your Spring Boot app fails to start after deployment. What steps do you take to resolve it?

✅ Answer:

Check application logs for stack trace.

Validate:

Port conflicts

Missing environment variables

Database connectivity

Incorrect configuration files

Roll back to previous stable version if needed.

Use health checks and readiness probes in containerized environments.

🧮 Scenario 13: High Load Handling

❓ Your API is experiencing high traffic. How do you ensure it scales effectively?

✅ Answer:

Horizontal scaling: Deploy multiple instances behind a load balancer.

Caching: Use Redis or in-memory cache for frequent reads.

Database optimization: Indexing, connection pooling.

Rate limiting: Prevent abuse using tools like Bucket4j.

Async processing: Offload heavy tasks to background jobs.

🧪 Scenario 14: Testing a Payment Flow

❓ You’re building a payment API. How do you test it safely?

✅ Answer:

Use sandbox environments provided by payment gateways.

Mock external APIs using WireMock or Mockito.

Write integration tests for end-to-end flow.

Validate edge cases: failed payments, retries, duplicate transactions.

🧹 Types of Garbage Collectors in Java

GC Type Description Best For

Serial GC Single-threaded collector. Simple and efficient for small applications. Small apps or single-core systems

Parallel GC Uses multiple threads for GC. Focuses on throughput. Multi-core systems, batch jobs

CMS (Concurrent Mark-Sweep) Minimizes pause time by doing most GC work concurrently. Deprecated in Java 14. Low-latency apps (pre-Java 14)

G1 GC (Garbage First) Breaks heap into regions. Prioritizes regions with most garbage. Large heaps, balanced performance

ZGC (Z Garbage Collector) Scalable, low-latency GC. Handles heaps up to terabytes with sub-millisecond pauses. Real-time systems, large heaps

Shenandoah GC Low-pause GC with concurrent compaction. Low-latency apps, large heaps

Epsilon GC No-op GC (doesn’t reclaim memory). Used for testing or short-lived apps. Performance testing only

🔍 Quick Comparison

Feature Serial GC Parallel GC G1 GC ZGC Shenandoah

Pause Time High Medium Low Very Low Very Low

Throughput Medium High High Medium Medium

Heap Size Support Small Medium Large Huge Large

Concurrent GC No Partial Yes Yes Yes

Java Version All All Default since Java 9 Java 11+ Java 12+

⚙️ How to Choose a GC

Small app or single-threaded → Serial GC

High throughput batch processing → Parallel GC

Balanced latency and throughput → G1 GC (default in Java 9+)

Ultra-low pause time → ZGC or Shenandoah

Testing memory behavior → Epsilon GC

🛠️ How to Set GC in JVM

You can specify the GC type using JVM flags:

bash

# G1 GC (default in Java 9+)

-XX:+UseG1GC

# Parallel GC

-XX:+UseParallelGC

# ZGC (Java 11+)

-XX:+UseZGC

# Shenandoah (Java 12+)

-XX:+UseShenandoahGC

Would you like help choosing the best GC for a specific type of application you're working on?

Solied Principle

Jenkins personal project m kha use krta h

If we call two multiple services and how we can set the priority?

Design Pattern and System Design?

[a,bc,def]

[f,ed,cba]

public class ReverseStrings {

public static void main(String[] args) {

List<String> input = Arrays.asList("a", "bc", "def");

List<String> output = new ArrayList<>();

for (String s : input) {

output.add(new StringBuilder(s).reverse().toString());

}

Collections.reverse(output);

System.out.println(output); // Output: [f, ed, cba]

}

}