## Spring Boot and Microservice Interview Questions

### 1) What is the main class in a Spring Boot application?

➔ The main class in a Spring Boot application is the entry point and is annotated with @SpringBootApplication

```
@SpringBootApplication
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

### 2) What does the @SpringBootApplication annotation do?

→ **@SpringBootApplication** is a convenience annotation that combines three annotations

**@Configuration** : (marks the class as a source of bean definitions)

**@EnableAutoConfiguration** : (enables Spring Boot's auto-configuration mechanism)

**@ComponentScan** : (scans the package of the annotated class for Spring components)

## 3) How do you handle exceptions in Spring Boot?

➔ You can handle exceptions in Spring Boot using @ControllerAdvice and @ExceptionHandler annotations to create a global exception handler.

```
@ControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<ErrorResponse> handleResourceNotFoundException(ResourceNo
        ErrorResponse errorResponse = new ErrorResponse("NOT_FOUND", ex.getMessage(
        return new ResponseEntity<>(errorResponse, HttpStatus.NOT_FOUND);
    }
}
```

## 4) What is Spring Boot Actuator and what are its benefits?

➔ Spring Boot Actuator provides production-ready features such as **health checks**, **metrics**, and **monitoring** for Spring Boot application.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

```
management.endpoints.web.exposure.include=health,info
```

## 5) What are Spring Profiles and how do you use them?

➔ Spring Profiles allow you to segregate parts of your application configuration and make it only available in certain environments.

➔ You can activate profiles using the **spring.profiles.active** property.

```
# application-dev.properties

spring.datasource.url=jdbc:mysql://localhost:3306/devdb

# application-prod.properties

spring.datasource.url=jdbc:mysql://localhost:3306/proddb
```

**6) How do you optimize the startup time of a Spring Boot application in a production environment?**

**Spring Boot provides various options for optimizing startup time. Key strategies include:**

**1) Lazy Initialization** : Use **spring.main.lazy-initialization=true** to delay bean initialization until needed.

2) **Profile-Specific Configuration** : Separate configurations per environment to avoid unnecessary loading.

3) **Component Scanning** : Restrict the scope of component scanning using @ComponentScan to only include essential packages.

4) **Reduce Bean Creation** : Avoid creating unnecessary beans during startup, especially for time-intensive services.

**By reducing the number of beans initialized upfront, you can significantly speed up application startup.**

**7) Explain the concept of Spring Boot's @ConfigurationProperties with complex objects. How would you handle nested configurations?**

- ➔ The **@ConfigurationProperties** annotation is a powerful way to map external configuration properties into Java objects
- ➔ For complex, nested configurations, Spring Boot can handle hierarchical properties through nested classes.

```java
@ConfigurationProperties(prefix = "app")
public class AppConfig {
    private Database database;
    private List<Service> services;

    public static class Database {
        private String url;
        private String username;
        private String password;
    }

    public static class Service {
        private String name;
        private int timeout;
    }
}
```

**8) What are the main challenges with distributed tracing in Spring Boot microservices, and how do you implement it?**

➔ Distributed tracing allows tracking requests across multiple microservices.
➔ The challenges include latency, proper correlation of requests, and aggregating trace data across services.

**Solution**:

- **Spring Cloud Sleuth**: Automatically instruments Spring Boot applications for distributed tracing.
- **Integration with Zipkin or Jaeger**: Use Sleuth with tools like Zipkin for trace visualization and monitoring.
- **Correlation**: Propagate TraceId and SpanId headers for cross-service correlation, ensuring traceability.

**9) What is Spring Boot's @Retryable annotation, and how do you fine-tune it for microservices reliability?**

➔ The @Retryable annotation in Spring Boot allows retrying a method call in case of failure.
➔ This is essential for improving the reliability of services that might experience transient failures (e.g., network timeouts or database issues).

```
@Retryable(value = {IOException.class}, maxAttempts = 5, backoff = @Backoff(delay =
public String fetchData() {
    // API call that might fail
}
```

**10) How do you handle versioning in Spring Boot APIs?**

**API versioning is crucial for maintaining backward compatibility as your services evolve. Common strategies include:**

**URI Versioning** : /api/v1/resource
**Parameter Versioning** : /api/resource?version=1
**Header-based Versioning** : Through custom headers like API-Version.

**11) How do you manage external configurations in a Spring Boot application across multiple environments?**

- **Profiles:** Use `@Profile` to define beans for specific environments (e.g., `@Profile("prod")`).
- **application.properties or YAML:** Use `application-prod.properties` for production-specific configurations.
- **Spring Cloud Config:** For distributed systems, use Spring Cloud Config Server to manage configurations centrally.

## 12) What are some common performance bottlenecks in Spring Boot applications and how do you resolve them?

### 1. Database Access

- ✅ **Optimize Queries:** Ensure SQL queries are efficient and use proper indexes.
- 📄 **Pagination:** Fetch data in chunks using pagination to avoid memory overload.
- 🔗 **Connection Pooling:** Leverage tools like **HikariCP** for efficient database connection management.

---

### 2. Memory Management

- 🧠 **Avoid Memory Leaks:** Regularly review your code for unclosed resources or lingering references.
- 🔍 **Monitoring:** Use tools like **VisualVM** or **JVisualVM** to profile memory usage and identify leaks.

---

### 3. Thread Pooling

- ⚙️ **Thread Pool Sizing:** Tune thread pools based on system resources and expected load.
- 🧵 **Executor Services:** Use Java's ExecutorService or Spring's @Async with properly configured task executors for managing background tasks and HTTP request handling.
-

---

### 4. Caching

- 🚀 **Spring Caching Abstraction:** Use annotations like @Cacheable, @CachePut, and @CacheEvict to reduce database hits.
- 🗃️ **Cache Providers:** Integrate with **Ehcache**, **Caffeine**, or **Redis** for high-performance caching solutions.

## 13) How do you configure and manage Spring Boot logging in production?

### 1. 🔊 Logback (Default Logging Framework)

- ✅ **Out-of-the-Box Support:** Spring Boot uses **Logback** as the default logging implementation.
- ⚙️ **Custom Configuration:** Configure advanced logging using logback-spring.xml for environment-specific setups.

### 2. 🌐 External Logging Integration

- 📦 **Centralized Logging:** Integrate with logging stacks like ELK (Elasticsearch + Logstash + Kibana) or EFK (Fluentd instead of Logstash).
- 📡 **Benefits:** Enables real-time log aggregation, analysis, and visualization across distributed services.

### 3. 🎚️ Log Levels per Environment

- 🔧 **Environment-Specific Tuning:**
  - DEBUG – for development to get detailed application flow.
  - INFO – for production to reduce noise but keep essential logs.

- 📝 **Configuration:** Set log levels in application.yml or application.properties:

## 14) How can you disable a specific auto-configuration class in Spring Boot ?

➔ Disable an auto-configuration class by adding the exclude attribute:

```
@SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

```
@EnableAutoConfiguration(exclude = { DataSourceAutoConfiguration.class })
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

## 15) How to configure multiple datasource in spring boot application ?

➔ Define Properties in application.yml or application.properties .

```yaml
spring:
  datasource:
    primary:
      url: jdbc:mysql://localhost:3306/primary_db
      username: root
      password: password
      driver-class-name: com.mysql.cj.jdbc.Driver

    secondary:
      url: jdbc:h2:mem:secondary_db
      driver-class-name: org.h2.Driver
      username: sa
      password:
```

## ➔ Configure primary datasource

```java
@Configuration
@EnableTransactionManagement
@EnableJpaRepositories(
    basePackages = "com.example.repo.primary",
    entityManagerFactoryRef = "primaryEntityManagerFactory",
    transactionManagerRef = "primaryTransactionManager"
)
public class PrimaryDataSourceConfig {

    @Bean
    @Primary
    @ConfigurationProperties("spring.datasource.primary")
    public DataSource primaryDataSource() {
        return DataSourceBuilder.create().build();
    }

    @Primary
    @Bean
    public LocalContainerEntityManagerFactoryBean primaryEntityManagerFactory(
            EntityManagerFactoryBuilder builder) {
        return builder
                .dataSource(primaryDataSource())
                .packages("com.example.entity.primary")
                .persistenceUnit("primary")
                .build();
    }

    @Primary
    @Bean
    public PlatformTransactionManager primaryTransactionManager(
            EntityManagerFactory primaryEntityManagerFactory) {
        return new JpaTransactionManager(primaryEntityManagerFactory);
    }
}
```

## Configure secondary datasource

```java
@Configuration
@EnableJpaRepositories(
    basePackages = "com.example.repo.secondary",
    entityManagerFactoryRef = "secondaryEntityManagerFactory",
    transactionManagerRef = "secondaryTransactionManager"
)
public class SecondaryDataSourceConfig {

    @Bean
    @ConfigurationProperties("spring.datasource.secondary")
    public DataSource secondaryDataSource() {
        return DataSourceBuilder.create().build();
    }

    @Bean
    public LocalContainerEntityManagerFactoryBean secondaryEntityManagerFactory(
            EntityManagerFactoryBuilder builder) {
        return builder
                .dataSource(secondaryDataSource())
                .packages("com.example.entity.secondary")
                .persistenceUnit("secondary")
                .build();
    }

    @Bean
    public PlatformTransactionManager secondaryTransactionManager(
            EntityManagerFactory secondaryEntityManagerFactory) {
        return new JpaTransactionManager(secondaryEntityManagerFactory);
    }
}
```

### Key Points

- Use `@Primary` to mark one datasource as the default.
- Define separate `basePackages` for `repositories` and `entities` for each data source.
- Each data source must have its own:
  - `DataSource`
  - `EntityManagerFactory`
  - `TransactionManager`

## 16) Can you explain the purpose of Stereotype annotations in the Spring Framework ?

➔ Stereotype annotations in Spring are specialized annotations used to declare Spring-managed components.
➔ They help Spring automatically detect and register beans during component scanning.

### 🧠 Why Use Stereotype Annotations?

1. ✅ **Enable Component Scanning**
   - Classes annotated with stereotypes are automatically discovered and registered as beans.
2. ✅ **Improve Code Semantics**
   - Using `@Service`, `@Repository`, and `@Controller` makes your code **more readable and meaningful** by indicating the role of the class.
3. ✅ **Enable Aspect-Oriented Features**
   - For example, `@Repository` enables **exception translation** using Spring's `PersistenceExceptionTranslationPostProcessor`.
4. ✅ **Reduce Boilerplate Configuration**
   - No need for explicit `<bean>` declarations in XML or manual `@Bean` registration in config classes.

## 17) How many ways we can perform dependency injection in spring or spring boot ?

### ✅ 1. Constructor-Based Injection (Recommended ✅)

🔹 Dependencies are injected via the class constructor.

```java
@Service
public class OrderService {
    private final PaymentService paymentService;

    @Autowired
    public OrderService(PaymentService paymentService) {
        this.paymentService = paymentService;
    }

    // Or even better (since Spring 4.3+), no need for @Autowired
    public OrderService(PaymentService paymentService) {
        this.paymentService = paymentService;
    }
}
```

🟢 **Benefits:**

- Promotes immutability
- Easy for unit testing
- Preferred for mandatory dependencies

### ✅ 2. Setter-Based Injection

🔹 Dependencies are injected via setter methods.

```java
@Service
public class OrderService {

    private PaymentService paymentService;

    @Autowired
    public void setPaymentService(PaymentService paymentService) {
        this.paymentService = paymentService;
    }
}
```

🟡 **Use Case:**

- When the dependency is **optional** or needs to be set **after construction**

### ✅ 3. Field-Based Injection (Not recommended ❌)

🔹 Dependencies are injected directly into fields.

```java
@Service
public class OrderService {

    @Autowired
    private PaymentService paymentService;
}
```

🔴 **Drawbacks:**

- Harder to test (no constructor for mocking)
- Violates principles of encapsulation
- Not ideal for required dependencies

## 18) Can you provide an example of a real-world use case where @PostConstruct is particularly useful?

### ➔ Preloading Data on Application Startup

```java
@Service
public class ProductCacheService {

    private final ProductRepository productRepository;
    private Map<Long, Product> productCache = new HashMap<>();

    public ProductCacheService(ProductRepository productRepository) {
        this.productRepository = productRepository;
    }

    @PostConstruct
    public void initCache() {
        List<Product> products = productRepository.findAll();
        products.forEach(product -> productCache.put(product.getId(), product));
        System.out.println("✅ Product cache initialized with " + products.size() + " items");
    }

    public Product getProductById(Long id) {
        return productCache.get(id);
    }
}
```

#### 💬 What Happens Here?

- Spring injects the `ProductRepository`.
- After the bean is fully constructed and dependencies injected, `@PostConstruct` runs.
- The `initCache()` method loads all product data into a local cache for faster access.

#### 🧠 Why Use `@PostConstruct` Instead of a Constructor?

- You **can't call other Spring beans** safely in a constructor.
- `@PostConstruct` ensures the bean is **fully initialized** before the method runs.

#### ✅ Summary

| Use Case | Why `@PostConstruct` Helps |
|---|---|
| Load data into cache | Waits until all dependencies are ready |
| Register with external services | Avoids initialization race conditions |
| Validate bean configuration | Ensures consistent startup checks |
| Trigger scheduled jobs manually | Pre-load cron logic if needed |

## 19) How can we dynamically load values in a Spring Boot application?

### ✅ 1. Using @Value Annotation (for simple values)

```java
@Component
public class AppInfo {

    @Value("${app.name}")
    private String appName;

    public void printAppName() {
        System.out.println("App Name: " + appName);
    }
}
```

### ✅ 2. Using @ConfigurationProperties (for grouped values)

yaml

```yaml
app:
  config:
    name: Inventory System
    version: 1.0
    enabled: true
```

java

```java
@Component
@ConfigurationProperties(prefix = "app.config")
public class AppConfig {
    private String name;
    private String version;
    private boolean enabled;

    // Getters and setters
}
```

## ✅ 3. Environment Object (for runtime value resolution)

```java
@Component
public class DynamicLoader {

    @Autowired
    private Environment env;

    public void load() {
        String dbUser = env.getProperty("spring.datasource.username");
        System.out.println("Database User: " + dbUser);
    }
}
```

🟢 Good for runtime evaluation or conditional config logic.

## ✅ 4. External Configuration (via Environment Variables or CLI)

```
java -jar app.jar --server.port=8085 --app.name="Staging App"
```

## ✅ 5. Dynamic Loading from a Database / Remote Source

➔ You can fetch values at runtime (e.g., for feature flags or settings) using @PostConstruct, a scheduled task, or an external config service.

```
@Service
public class ConfigService {

    private final ConfigRepository configRepository;

    private Map<String, String> dynamicConfig = new HashMap<>();

    public ConfigService(ConfigRepository configRepository) {
        this.configRepository = configRepository;
    }

    @PostConstruct
    public void loadFromDb() {
        List<ConfigEntity> configs = configRepository.findAll();
        configs.forEach(c -> dynamicConfig.put(c.getKey(), c.getValue()));
    }

    public String getValue(String key) {
        return dynamicConfig.getOrDefault(key, "N/A");
    }
}
```

## 20) How to load External Properties in Spring Boot ?

➔ **Loading external properties in a Spring Boot application is a common way to separate configuration from code — especially useful for different environments (dev, test, prod).**

### ✅ 1. Use --spring.config.location Command-Line Argument

```
java -jar myapp.jar --spring.config.location=file:/path/to/config.properties
```

### ✅ 2. Use SPRING_CONFIG_LOCATION Environment Variable

```
export SPRING_CONFIG_LOCATION=file:/path/to/config/
java -jar myapp.jar
```

🟢 Useful in cloud or containerized environments (e.g., Kubernetes, Docker).

### ✅ 3. Use @PropertySource in Java Config (for custom file names)

```
@Configuration
@PropertySource("file:/external/config/custom-config.properties")
public class ExternalConfig {
}
```

- Works with `.properties`, but **not** `.yml`
- Use `Environment` or `@Value` to access values

## ✅ 4. Use External Config in Spring Boot Profiles

Structure external configs per environment:

arduino Copy

```
/config/
  ├── application.yml
  ├── application-dev.yml
  └── application-prod.yml
```

Then activate the profile:

bash Copy

```
java -jar app.jar --spring.profiles.active=dev --spring.config.location=file:/config/
```

## 21) Can we avoid this dependency ambiguity without using @Qualifier ?

- 1. **Use @Primary to Designate a Default Bean**

```
@Bean
@Primary
public NotificationService emailNotificationService() {
    return new EmailNotificationService();
}

@Bean
public NotificationService smsNotificationService() {
    return new SmsNotificationService();
}
```

- 2. **Inject by Concrete Class Instead of Interface**

```
@Autowired
private SmsNotificationService smsNotificationService;
```

- 3. **Use @ComponentScan Selectively**

Avoid registering multiple beans of the same type by refining your @ComponentScan.

```
@SpringBootApplication
@ComponentScan(basePackages = "com.example.email") // excludes other implementations
public class MyApp {}
```

- 4. **Use Profiles to Load Only One Bean at a Time**

```java
@Bean
@Profile("email")
public NotificationService emailNotificationService() {
    return new EmailNotificationService();
}

@Bean
@Profile("sms")
public NotificationService smsNotificationService() {
    return new SmsNotificationService();
}
```

```
--spring.profiles.active=email
```

## 22) What is bean scope & Can you explain different type of bean scope ?

### ✅ Common Types of Bean Scopes in Spring

Spring supports several scopes, especially in different environments (web vs. non-web).

| Scope | Description | Applicable In |
|---|---|---|
| `singleton` | Only **one instance** per Spring container (default) | Core, Web |
| `prototype` | **New instance** each time it is requested | Core, Web |
| `request` | One instance per **HTTP request** | Web (Servlet) |
| `session` | One instance per **HTTP session** | Web (Servlet) |
| `application` | One instance per **ServletContext (application-wide)** | Web (Servlet) |
| `websocket` | One instance per **WebSocket session** | WebSocket apps |

## 23) Can you provide a few real-time use cases for when to choose Singleton scope and Prototype scope ?

### 🧠 Summary Comparison

| Aspect | Singleton Scope | Prototype Scope |
|---|---|---|
| Instance Count | One per Spring container | New every time it's requested |
| Lifecycle Managed | Fully by Spring | Only initialization (no destroy callback) |
| Use For | Stateless, shared, thread-safe beans | Stateful, non-thread-safe, custom state |
| Examples | Service, Repository, Config, Caching | Report builder, Upload job, Worker task |

## 24) How can we deserialize a JSON request payload into an object ?

➔ **Jackson**

## 25) How can we perform content negotiation (XML/JSON) in Rest endpoint ?

### 1. Using HTTP Headers (Most common way)

**Client sends:**

- `Accept` **header** indicating the desired response format.
  - Example:
    - `Accept: application/json`
    - `Accept: application/xml`
    - Or `Accept: application/json, application/xml;q=0.8` (priority-based)

**Server responds:**

- Examines the `Accept` header.
- Returns the response in the requested format.
- Sets `Content-Type` header accordingly ( `application/json` or `application/xml` ).

### 2. How to implement in popular frameworks

**In Spring Boot (Java)**

- Just add support for both JSON and XML in your `pom.xml` or `build.gradle` (Jackson for JSON and JAXB or Jackson XML for XML).
- Use `@RestController` and simply return the POJO.
- Spring Boot automatically negotiates content based on the `Accept` header.

## 26) How do you maintain the versioning for your REST API?

### 1. URI Path Versioning (Most common & explicit)

Add the version in the URL path:

```bash
/api/v1/users
/api/v2/users
```

### 2. Request Header Versioning

Use a custom header (e.g., `Accept` or `X-API-Version`):

```bash
GET /api/users
Accept: application/vnd.myapp.v1+json
```

### 3. Query Parameter Versioning

Pass version as a query param:

```pgsql
GET /api/users?version=1
```

## 27) How will you document your rest API ?

### Ways to Document REST APIs

#### 1. Use OpenAPI (Swagger) Specification

- The industry standard for REST API documentation.
- Describes endpoints, request/response formats, status codes, authentication, etc.
- Can generate interactive documentation (Swagger UI) where users can try out API calls.

#### 2. Auto-generate Docs From Code Annotations

- Use tools that scan your code and produce OpenAPI specs automatically.
- Examples:
  - **Spring Boot:** Use `springdoc-openapi` or `springfox-swagger2`.
  - **Express:** Use `swagger-jsdoc` and `swagger-ui-express`.
  - **Flask:** Use `flasgger` or `apispec`.

#### 3. Write Manual Documentation

- Create human-readable docs that explain:
  - Available endpoints and their HTTP methods
  - Required/request parameters and body
  - Response formats and status codes
  - Error messages
  - Authentication/authorization
- Can be done in markdown, wikis, or API portals like **ReadMe**, **Postman**, or **GitHub Pages**.

## 28) How will you consume restful API ?

### Java (Spring Boot with `RestTemplate` or `WebClient`)

Using `RestTemplate` (Legacy)

```java
RestTemplate restTemplate = new RestTemplate();
String url = "https://api.example.com/users/1";
User user = restTemplate.getForObject(url, User.class);
```

Using `WebClient` (Preferred for reactive & modern apps)

```java
WebClient webClient = WebClient.create();

User user = webClient.get()
        .uri("https://api.example.com/users/1")
        .retrieve()
        .bodyToMono(User.class)
        .block(); // blocking call for synchronous processing
```

## 29 ) How can you hide certain REST endpoints to prevent them from being exposed externally?

### 2. Do Not Document Them in OpenAPI/Swagger

Exclude endpoints from appearing in your Swagger UI or OpenAPI spec.

Spring Boot Example with Springdoc OpenAPI:

```java
@Hidden
@GetMapping("/api/internal/debug")
public String debugEndpoint() {
    return "Internal info";
}
```

### 1. Use Spring Security (or any auth framework)

Restrict access to internal endpoints by role or IP.

Spring Boot Example:

```java
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
        .antMatchers("/api/internal/**").hasRole("ADMIN") // Only ADMIN can access
        .antMatchers("/api/public/**").permitAll(); // Publicly accessible
}
```

### 5. Use Profiles or Feature Flags

Only expose certain controllers or endpoints in specific environments (e.g., d

Spring Boot Example:

```java
@Profile("dev")
@RestController
@RequestMapping("/api/internal")
public class InternalController {
    // only available in dev profile
}
```

## 30 ) How can you avoid defining handlers for multiple exceptions, or what is the best practice for handling exceptions ?

➔ Instead of writing try-catch blocks or separate handlers in every controller , use a **centralized exception handler** to handle all exceptions in one place.

```java
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<ErrorResponse> handleResourceNotFound(ResourceNotFoundException ex) {
        ErrorResponse error = new ErrorResponse(
            HttpStatus.NOT_FOUND.value(),
            ex.getMessage(),
            LocalDateTime.now()
        );
        return new ResponseEntity<>(error, HttpStatus.NOT_FOUND);
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErrorResponse> handleGenericException(Exception ex) {
        ErrorResponse error = new ErrorResponse(
            HttpStatus.INTERNAL_SERVER_ERROR.value(),
            "An unexpected error occurred",
            LocalDateTime.now()
        );
        return new ResponseEntity<>(error, HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

```java
public class ErrorResponse {
    private int status;
    private String message;
    private LocalDateTime timestamp;

    // constructors, getters, setters
}
```

✅ **Benefits of Centralized Exception Handling:**

- No repetitive try-catch blocks.
- Cleaner controller code.
- Consistent error response format.
- Easier to add logging, monitoring, alerts.

- Group similar exceptions if needed:

```java
@ExceptionHandler({IllegalArgumentException.class, IllegalStateException.class})
public ResponseEntity<?> handleBadRequest(RuntimeException ex) { ... }
```

## 31 ) How will you validate or sanitise your input payload ?

### 1. Use Bean Validation

Use annotations like @NotNull, @Email, @Size, etc., in your DTOs with javax.validation or jakarta.validation.

```java
public class UserRequest {

    @NotBlank(message = "Name is required")
    private String name;

    @Email(message = "Email must be valid")
    private String email;

    @Min(value = 18, message = "Age must be at least 18")
    private int age;

    // getters and setters
}
```

✅ In Controller:

```java
@PostMapping("/users")
public ResponseEntity<?> createUser(@Valid @RequestBody UserRequest request)
    // Validated input
    return ResponseEntity.ok("User created");
}
```

### 2. Handle Validation Errors Gracefully

Use `@RestControllerAdvice` to return consistent messages.

```java
@ExceptionHandler(MethodArgumentNotValidException.class)
public ResponseEntity<?> handleValidationErrors(MethodArgumentNotValidExceptio
    Map<String, String> errors = new HashMap<>();
    ex.getBindingResult().getFieldErrors().forEach(error ->
        errors.put(error.getField(), error.getDefaultMessage()));
    return ResponseEntity.badRequest().body(errors);
}
```

### 3. Sanitize Input (Optional, Based on Risk)

Sanitization is usually about cleaning or escaping input before using it (e.g., in UI or SQL

- Use frameworks (Hibernate, Spring Data JPA) to prevent SQL injection.
- For strings, you can strip/escape harmful HTML or script content.

**Example: Preventing XSS (if rendering user input in UI)**

```java
String cleanInput = HtmlUtils.htmlEscape(userInput);
```

## 32) How can you populate validation error message to the end users ?

➔ To **populate validation error messages to end users**, especially in REST APIs (like with Spring Boot), you need to:

1) **Use validation annotations** in your DTO.
2) **Catch validation errors** centrally.
3) **Return a structured, user-friendly error response**.

### 🔹 1. Add Validation Annotations in DTO

```
public class UserRequest {

    @NotBlank(message = "Name is required")
    private String name;

    @Email(message = "Please provide a valid email")
    private String email;

    @Min(value = 18, message = "Age must be at least 18")
    private int age;

    // getters and setters
}
```

### 2. Use @Valid in controller

```
@PostMapping("/users")
public ResponseEntity<?> createUser(@Valid @RequestBody UserRequest request) {
    // Only called if validation passes
    return ResponseEntity.ok("User created");
}
```

### 3 . Handle validation with @ControllerAdvice

```java
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<Map<String, String>> handleValidationErrors(MethodArgumentNotValidExcept
        Map<String, String> errors = new HashMap<>();
        ex.getBindingResult().getFieldErrors().forEach(error ->
            errors.put(error.getField(), error.getDefaultMessage())
        );
        return new ResponseEntity<>(errors, HttpStatus.BAD_REQUEST);
    }
}
```

## 🧠 Best Practices

| Practice | Benefit |
|---|---|
| Use custom error messages | More user-friendly |
| Centralize validation handling | DRY code and consistent error format |
| Return clear field-error mapping | Helps frontend show exact field issues |
| Localize messages (optional) | Support for international users |

## 🌐 Bonus: Localization Support

Add to `messages.properties`:

```properties
user.email.invalid=Please provide a valid email
user.name.blank=Name is required
```

Then update annotations:

```java
@Email(message = "{user.email.invalid}")
@NotBlank(message = "{user.name.blank}")
```

**33) Let's say you find a bug in production environment and now you want to debug that scenario ,How can you do that from your local ?**

➔ **Debugging a bug found in the production environment from your local can be tricky.**

### 🔹 1. Understand and Reproduce the Bug

- **Gather details** from logs, bug reports, monitoring tools, or support tickets:
  - Error messages
  - Request payloads
  - Stack traces
  - Affected users
  - Timestamp of issue
- **Ask**: Can the issue be reproduced reliably?

### 🔹 2. Check Production Logs / Monitoring Tools

Use observability tools such as:

- 🔍 **ELK Stack** (Elasticsearch, Logstash, Kibana)
- 📊 **Prometheus + Grafana**
- 📁 Log files via CloudWatch (AWS), Stackdriver (GCP), or other logging systems.

Get the exact:

- Input/request data
- Headers
- Auth tokens (if applicable)
- DB state (snapshot or relevant rows)

### 🔹 3. Mimic Production Environment Locally

- **Checkout the same Git branch or tag** deployed to prod.
- Configure the same:
  - Environment variables
  - App configs (like external services, feature flags)
- Use **Docker / Docker Compose / Kubernetes manifests** to simulate the same infrastructure locally.

### 🔹 4. Replay the Bug Locally

- Use the **exact same input** from production (e.g., REST API request).
- Tools:
  - Postman (to replicate requests)
  - `curl` with prod payloads
  - JUnit/Integration tests with prod data
- Simulate edge cases like:
  - Null/missing fields
  - Invalid states in DB

### 🔹 5. **Attach Debugger**

In your IDE:

- Set breakpoints around the suspect code.
- Run the app in **debug mode** (`debug` profile or IDE debug button).
- Step through the flow using actual data that caused the issue.

### 🔹 6. **Write Failing Test Case (TDD-style Debugging)**

Write a unit or integration test using the same input and DB state. This helps:

- Isolate the problem.
- Prevent future regressions (part of automated test suite).

java Copy Edit

```java
@Test
void shouldThrowException_WhenInputIsInvalid() {
    // simulate production scenario
}
```

### 🔹 7. **Fix the Bug & Retest Locally**

- Once identified, fix the issue.
- Rerun the test.
- Run full test suite (unit + integration + contract tests).

### 🔹 8. **Peer Review + Deploy Safely**

- Create a **pull request** and request **code review**.
- Deploy to staging or QA first, verify the fix.
- Use **feature toggles** or **canary releases** to roll out gradually if needed.

### 🔹 9. **Monitor After Deployment**

- Watch logs and alerts post-deployment.
- Confirm the issue no longer occurs.
- Possibly add observability (metrics, logs) in fixed area.

## 34 ) What is the key benefit of using the Spring Data JPA ?

### 1. 🚀 Reduces Boilerplate Code

- No need to write repetitive CRUD logic.
- You get basic methods like `save()`, `findById()`, `findAll()`, `delete()` out of the box

java

```
public interface UserRepository extends JpaRepository<User, Long> {}
```

### 2. 🧠 Powerful Query Generation from Method Names

- You can define queries just by naming methods in a certain way.

java

```
List<User> findByEmail(String email);
List<User> findByFirstNameAndLastName(String first, String last);
```

Spring automatically translates these into SQL queries.

### 3. 📜 Supports JPQL and Native Queries

- When custom logic is needed, you can still write JPQL or native SQL.

java

```
@Query("SELECT u FROM User u WHERE u.age > :age")
List<User> findUsersOlderThan(@Param("age") int age);
```

### 4. 📦 Pagination and Sorting Support

- Built-in support for paging and sorting results.

java

```
Page<User> findAll(Pageable pageable);
List<User> findAll(Sort sort);
```

### 9. 🔄 Repository Customization

- You can define your own methods alongside Spring-generated ones.
- Supports custom repository implementation for complex logic.

## 35) What are the different ways to define custom queries in Spring Data JPA ?

### 1. Derived Query Methods (Query by Method Name)

Let Spring generate the SQL based on method name.

```java
List<User> findByFirstNameAndStatus(String firstName, String status);
```

✔ Simple and readable, but limited for complex queries.

### 2. JPQL with `@Query` Annotation

Use JPQL (object-oriented SQL) to define custom queries.

```java
@Query("SELECT u FROM User u WHERE u.status = :status")
List<User> findActiveUsers(@Param("status") String status);
```

✔ Uses entity field names, not table/column names.

### 3. Native SQL with `@Query(nativeQuery = true)`

When you need full SQL power (joins, DB-specific functions).

```java
@Query(value = "SELECT * FROM users WHERE email = :email", nativeQuery = true)
User findByEmailNative(@Param("email") String email);
```

✔ Useful when JPQL doesn't support required operations.

### 4. Named Queries (Defined in Entity)

Define queries inside entity using `@NamedQuery`.

```java
@Entity
@NamedQuery(name = "User.findByEmail", query = "SELECT u FROM User u WHERE u.email = :email")
public class User { ... }

interface UserRepository extends JpaRepository<User, Long> {
    User findByEmail(@Param("email") String email); // Will use the named query
}
```

✔ Good for reuse across multiple places, but harder to maintain.

### 🔹 6. Dynamic Queries with `Specification<T>` (Criteria API)

Use `JpaSpecificationExecutor<T>` for building dynamic queries at runtime.

```java
public class UserSpecs {
    public static Specification<User> hasStatus(String status) {
        return (root, query, cb) -> cb.equal(root.get("status"), status);
    }
}
```

Usage:

```java
userRepository.findAll(UserSpecs.hasStatus("ACTIVE"));
```

> ✔ Best for dynamic search filters.

### 🔹 7. Custom Repository Implementation

When logic is too complex for annotations or specifications.

java Copy Edit

```java
public interface UserRepositoryCustom {
    List<User> customSearchLogic(String keyword);
}

public class UserRepositoryImpl implements UserRepositoryCustom {
    @PersistenceContext
    private EntityManager em;

    @Override
    public List<User> customSearchLogic(String keyword) {
        return em.createQuery("SELECT u FROM User u WHERE u.name LIKE :kw", User.class)
                 .setParameter("kw", "%" + keyword + "%")
                 .getResultList();
    }
}
```

## 36) Relationship mapping ?

### 🧠 Summary Table

| Relationship Type | Annotation | Example Entities | Notes |
|---|---|---|---|
| One-to-One | `@OneToOne` | User → Profile | Use `@JoinColumn` |
| One-to-Many | `@OneToMany` | Department → Employees | Use `mappedBy` on parent side |
| Many-to-One | `@ManyToOne` | Employee → Department | Child owns the FK |
| Many-to-Many | `@ManyToMany` | Student ↔ Course | Uses a join table |

### 🔹 1. Customer Entity (Parent)

java Copy

```java
@Entity
public class Customer {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String email;

    @OneToMany(mappedBy = "customer", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Order> orders = new ArrayList<>();

    // Getters and Setters
```

### 🔹 2. Order Entity (Child)

java

```java
@Entity
@Table(name = "orders")  // optional: avoids SQL reserved keyword conflict
public class Order {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private LocalDate orderDate;
    private Double totalAmount;

    @ManyToOne
    @JoinColumn(name = "customer_id")  // foreign key in orders table
    private Customer customer;

    // Getters and Setters
}
```

## 🧠 Explanation of Key Points

| Concept | Applied On | Explanation |
|---|---|---|
| `@OneToMany` | `Customer.orders` | One customer has many orders |
| `mappedBy = "customer"` | Tells JPA that `Order.customer` owns the relationship | |
| `@ManyToOne` | `Order.customer` | Many orders can belong to one customer |
| `@JoinColumn` | On `Order.customer` | Adds a `customer_id` column in `orders` table |
| `CascadeType.ALL` | On `Customer.orders` | Saves/deletes orders when customer is saved/deleted |
| `orphanRemoval = true` | On `Customer.orders` | Deletes orders removed from the list in code |

**37 ) Is this possible to execute Join query in Spring Data JPA ? If yes, how can you add some insights ?**

### 🔹 1. JPQL Join (Preferred for Entity Relationships)

📄 **Example: Join `Customer` and `Order`**

Assuming:

- A `Customer` entity has `@OneToMany` Orders
- An `Order` has `@ManyToOne` Customer

```java
@Query("SELECT o FROM Order o JOIN o.customer c WHERE c.email = :email")
List<Order> findOrdersByCustomerEmail(@Param("email") String email);
```

> 🔍 This will fetch orders placed by a customer with the specified email.

### 🔹 2. Join with Fetch (To Avoid N+1 Problem)

Fetch joins load associated entities in a single query.

```java
@Query("SELECT c FROM Customer c JOIN FETCH c.orders WHERE c.id = :id")
Customer findCustomerWithOrders(@Param("id") Long id);
```

> ⚡ Improves performance by preventing multiple SQL queries.

### 🔹 3. Native SQL Join (When You Need Table/Column Control)

You can write raw SQL if needed.

```java
@Query(value = "SELECT o.* FROM orders o INNER JOIN customer c ON o.customer_id
List<Order> findOrdersByCustomerEmailNative(@Param("email") String email);
```

> ✅ Best when you need full control over complex joins or use SQL-specific features.

## 38 ) How will you implement pagination & Sorting in Spring Data JPA ?

### 🔹 1. Repository Setup

Your repository must extend `JpaRepository` (or `PagingAndSortingRepository`):

```java
public interface ProductRepository extends JpaRepository<Product, Long> {
    Page<Product> findByCategory(String category, Pageable pageable);
}
```

✅ `Pageable` enables pagination & sorting in a single argument.

### 🔹 2. Using `Pageable` in Service or Controller

➤ Paginate all products (with sorting)

```java
Pageable pageable = PageRequest.of(pageNumber, pageSize, Sort.by("price").descending());
Page<Product> pagedResult = productRepository.findAll(pageable);
```

- `PageRequest.of(page, size)` → 0-based page index
- `Sort.by("field").direction()` → ascending or descending

### 🔹 3. Sorting Alone (without pagination)

```java
List<Product> products = productRepository.findAll(Sort.by("name").ascending());
```

### 🔧 Sorting on Multiple Fields

```java
Sort sort = Sort.by(Sort.Order.desc("price"), Sort.Order.asc("name"));
Pageable pageable = PageRequest.of(0, 10, sort);
```

### 🔄 Custom Queries with Pagination

```java
@Query("SELECT p FROM Product p WHERE p.category = :category")
Page<Product> findByCategory(@Param("category") String category, Pageable pageable);
```

✅ Works seamlessly with both JPQL and native queries.

**39) What is transaction management and how it works ? explain**

→ **Transaction Management in Spring (or any enterprise app) ensures that a set of operations either all succeed or all fail as one atomic unit of work.**

✅ **How Transaction Management Works in Spring**

➔ Spring provides declarative transaction management using the @Transactional annotation.

### 🔸 1. Using `@Transactional` Annotation

```java
@Service
public class OrderService {

    @Autowired
    private OrderRepository orderRepository;

    @Transactional
    public void placeOrder(Order order) {
        orderRepository.save(order);
        // if any exception occurs here, everything rolls back
    }
}
```

✅ **All DB operations within this method are part of one transaction. If any exception occurs, everything is rolled back automatically.**

## 🔸 2. Transactional Boundaries

Spring creates a **proxy** around your bean. When a method marked with `@Transactional` is called **from outside** the bean, Spring begins a transaction, manages it, and commits/rolls back based on outcome.

## 🔸 3. Transaction Rollback Rules

By default:

- **Unchecked exceptions** (`RuntimeException`, `Error`) → trigger rollback
- **Checked exceptions** → do **not** trigger rollback unless explicitly configured:

```java
@Transactional(rollbackFor = Exception.class)
```

## 🔸 4. Propagation Behavior

Defines **how a transaction behaves when a method is called from another transactional method.**

| Propagation | Meaning |
|---|---|
| `REQUIRED` (default) | Join existing or create a new one |
| `REQUIRES_NEW` | Always creates a new transaction |
| `NESTED` | Executes within a nested transaction |
| `MANDATORY` | Must run inside a transaction, or fail |
| `NOT_SUPPORTED` | Runs without a transaction |

## 🔸 5. Isolation Levels

Controls **visibility of data** between transactions.

| Level | Description |
|---|---|
| `READ_UNCOMMITTED` | Can read uncommitted changes (dirty read) |
| `READ_COMMITTED` | Only reads committed data |
| `REPEATABLE_READ` | Ensures same data is read again |
| `SERIALIZABLE` | Fully isolated (slowest but safest) |

Example:

```java
@Transactional(isolation = Isolation.READ_COMMITTED)
```

## 🧠 Real-Life Example

```java
@Transactional
public void transferMoney(Long fromAccount, Long toAccount, BigDecimal amount) {
    Account source = accountRepo.findById(fromAccount).get();
    Account target = accountRepo.findById(toAccount).get();

    source.debit(amount);
    target.credit(amount);

    accountRepo.save(source);
    accountRepo.save(target);
}
```

✅ If any step fails, no money is transferred.

## 40 ) Can we use transaction in private methods ?

➔ **No, you cannot use @Transactional effectively on private methods in Spring.**

## 41 ) How will you handle transaction in distributed microservice ?

➔ **Handling transactions in distributed microservices is one of the most challenging aspects of microservices architecture.**
➔ **Since each microservice has its own database, traditional ACID transactions don't work across services.**
➔ **Therefore, we use eventual consistency and patterns like**

### ✅ 1. Saga Pattern (Recommended)

The **Saga Pattern** manages distributed transactions using a sequence of **local transactions**. Each service performs its task and then publishes an event to trigger the next step.

#### 🔹 Two Types of Sagas:

| Type | Description |
|---|---|
| **Choreography** | No central coordinator. Services listen and react to events. |
| **Orchestration** | A central "orchestrator" tells services what to do next. |

### ✅ 3. Event-Driven Architecture (EDA)

Use **Kafka, RabbitMQ, or EventBridge** to emit events after local DB commits. Other services consume those events and act accordingly.

- Use **Outbox Pattern** to safely publish events after DB commits.
- Use **Change Data Capture (CDC)** with Debezium.

## 42) Explain orchestration and choreography pattern with example ?

### 🌀 1. Choreography-Based Saga (Event-Driven)

**Flow:**

**Each service listens to events and publishes the next event.**

**No central control.**

### ✅ Use Case: E-commerce Order Placement

**Microservices involved:**

1) **Order Service**
2) **Inventory Service**
3) **Payment Service**

- **Dependencies:**

- **Spring Boot**
- **Kafka (or RabbitMQ)**

### Sample Events (DTOs)

java

```java
public class OrderCreatedEvent {
    private Long orderId;
    private List<Long> productIds;
}

public class InventoryReservedEvent {
    private Long orderId;
}

public class PaymentCompletedEvent {
    private Long orderId;
}
```

### Order Service

java Copy Edit

```java
@Service
public class OrderService {

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;

    public void placeOrder(Order order) {
        // save order to DB
        kafkaTemplate.send("order-created", new OrderCreatedEvent(order.getId(), order.getProductId
    }
}
```

### Order Service (final completion)

java

```java
@KafkaListener(topics = "payment-completed")
public void markOrderCompleted(PaymentCompletedEvent event) {
    // update order status to COMPLETED
}
```

### Inventory Service

java Copy Edit

```java
@KafkaListener(topics = "order-created")
public void reserveInventory(OrderCreatedEvent event) {
    // Check & reserve inventory
    kafkaTemplate.send("inventory-reserved", new InventoryReservedEvent(event.getOrderId()));
}
```

### Payment Service

java Copy E

```java
@KafkaListener(topics = "inventory-reserved")
public void processPayment(InventoryReservedEvent event) {
    // Deduct money
    kafkaTemplate.send("payment-completed", new PaymentCompletedEvent(event.getOrderId()));
}
```

## 🧭 2. Orchestration-Based Saga

### Flow:

- A central **Orchestrator service** controls the steps and invokes each service.
- 💡 All services must expose APIs for compensate actions (cancel, release, refund).

Saga Orchestrator Logic:

java Copy

```java
@Service
public class SagaService {

    @Autowired OrderClient orderClient;
    @Autowired PaymentClient paymentClient;
    @Autowired InventoryClient inventoryClient;

    public void startOrderSaga(OrderRequest request) {
        try {
            orderClient.createOrder(request);
            paymentClient.processPayment(request);
            inventoryClient.deductStock(request);
            System.out.println("Order Saga Completed Successfully");
        } catch (Exception e) {
            System.out.println("Saga failed: " + e.getMessage());
            compensate(request);
        }
    }

    private void compensate(OrderRequest request) {
        try { inventoryClient.restoreStock(request); } catch (Exception ignored) {}
        try { paymentClient.refundPayment(request); } catch (Exception ignored) {}
        try { orderClient.cancelOrder(request); } catch (Exception ignored) {}
```

Expose a REST Endpoint:

java Copy

```java
@RestController
@RequestMapping("/saga")
public class SagaController {

    @Autowired SagaService sagaService;

    @PostMapping("/order")
    public ResponseEntity<String> createOrder(@RequestBody OrderRequest request) {
        sagaService.startOrderSaga(request);
        return ResponseEntity.ok("Saga Execution Started");
    }
}
```

## 2. 📦 Order Service

**Endpoints:**

java Copy

```java
@RestController
@RequestMapping("/orders")
public class OrderController {

    @PostMapping
    public ResponseEntity<String> createOrder(@RequestBody OrderRequest request) {
        System.out.println("Order Created: " + request.getOrderId());
        return ResponseEntity.ok("Order Created");
    }

    @PostMapping("/cancel")
    public ResponseEntity<String> cancelOrder(@RequestBody OrderRequest request) {
        System.out.println("Order Cancelled: " + request.getOrderId());
        return ResponseEntity.ok("Order Cancelled");
    }
}
```

## 3. 💳 Payment Service

**Endpoints:**

java Copy

```java
@RestController
@RequestMapping("/payments")
public class PaymentController {

    @PostMapping
    public ResponseEntity<String> processPayment(@RequestBody OrderRequest request) {
        System.out.println("Payment Processed for Order: " + request.getOrderId());
        return ResponseEntity.ok("Payment Processed");
    }

    @PostMapping("/refund")
    public ResponseEntity<String> refundPayment(@RequestBody OrderRequest request) {
        System.out.println("Payment Refunded for Order: " + request.getOrderId());
        return ResponseEntity.ok("Payment Refunded");
    }
}
```

## 4. 📦 Inventory Service

**Endpoints:**

java

Copy

```java
@RestController
@RequestMapping("/inventory")
public class InventoryController {

    @PostMapping("/deduct")
    public ResponseEntity<String> deductStock(@RequestBody OrderRequest request) {
        System.out.println("Stock Deducted: " + request.getProductId());
        // Simulate failure
        if ("fail".equals(request.getProductId())) {
            throw new RuntimeException("Stock not available");
        }
        return ResponseEntity.ok("Stock Deducted");
    }

    @PostMapping("/restore")
    public ResponseEntity<String> restoreStock(@RequestBody OrderRequest request) {
        System.out.println("Stock Restored: " + request.getProductId());
        return ResponseEntity.ok("Stock Restored");
    }
}
```

## 5. 🎯 Orchestrator Service

### Feign Clients:

java

```java
@FeignClient(name = "order-service")
public interface OrderClient {
    @PostMapping("/orders")
    void createOrder(@RequestBody OrderRequest request);

    @PostMapping("/orders/cancel")
    void cancelOrder(@RequestBody OrderRequest request);
}

@FeignClient(name = "payment-service")
public interface PaymentClient {
    @PostMapping("/payments")
    void processPayment(@RequestBody OrderRequest request);

    @PostMapping("/payments/refund")
    void refundPayment(@RequestBody OrderRequest request);
}
```


```java
@FeignClient(name = "payment-service")
public interface PaymentClient {
    @PostMapping("/payments")
    void processPayment(@RequestBody OrderRequest request);

    @PostMapping("/payments/refund")
    void refundPayment(@RequestBody OrderRequest request);
}
```


```java
@FeignClient(name = "inventory-service")
public interface InventoryClient {
    @PostMapping("/inventory/deduct")
    void deductStock(@RequestBody OrderRequest request);

    @PostMapping("/inventory/restore")
    void restoreStock(@RequestBody OrderRequest request);
}
```

## 43 ) Idempotency Situation ?

➔ Idempotency means that performing the same operation multiple times produces the same result as doing it once.

### 📌 Examples:

| Action | Idempotent? | Why? |
|---|---|---|
| `GET /orders/123` | ✅ Yes | Always returns the same order data. |
| `POST /orders` | ❌ No | Each call may create a new order (duplicate). |
| `DELETE /orders/123` | ✅ Yes | Deleting the same order repeatedly has the same effect. |

### 🛠️ Why is Idempotency Important?

In distributed systems (like microservices), **network retries**, **timeouts**, or **client retries** may cause the same request to be sent multiple times. Without idempotency:

- Duplicate entries might be created.
- Payments may be processed more than once.
- Data consistency may break.

➔ Make non-idempotent operations (like POST) **idempotent** by handling duplicate requests gracefully.

## 👩‍💻 How to Implement Idempotency in Spring Boot

### ✅ Strategy

Use an **Idempotency Key** (a unique request identifier) sent in the header or body. Before processing the request:

1. Check if a request with that key has already been processed.
2. If yes, **return the stored response**.
3. If not, **process it**, store the key and result.

### 🧱 Components Needed

- A table to store processed idempotency keys.
- A filter or interceptor to check the key before processing.
- Custom logic to cache and return previous responses.

## 📦 Step-by-Step Code

```
@Entity
public class IdempotencyRecord {
    @Id
    private String idempotencyKey;

    private String responseBody; // Optional: store serialized response
    private LocalDateTime createdAt;
}
```

```
public interface IdempotencyRecordRepository extends JpaRepository<IdempotencyRecord, String> {
}
```

```
@RestController
@RequestMapping("/orders")
public class OrderController {

    @Autowired
    private OrderService orderService;

    @PostMapping
    public ResponseEntity<String> createOrder(
            @RequestHeader("Idempotency-Key") String idempotencyKey,
            @RequestBody OrderRequest orderRequest) {

        return orderService.createOrder(orderRequest, idempotencyKey);
    }
}
```

```java
@Service
public class OrderService {

    @Autowired
    private IdempotencyRecordRepository idempotencyRepo;

    public ResponseEntity<String> createOrder(OrderRequest request, String key) {
        Optional<IdempotencyRecord> recordOpt = idempotencyRepo.findById(key);

        if (recordOpt.isPresent()) {
            // Already processed, return stored response
            return ResponseEntity.ok("Duplicate request. Order already created.");
        }

        // ✅ Process business logic
        System.out.println("Processing order: " + request.getOrderId());

        // Save idempotency record
        IdempotencyRecord record = new IdempotencyRecord();
        record.setIdempotencyKey(key);
        record.setCreatedAt(LocalDateTime.now());
        idempotencyRepo.save(record);

        return ResponseEntity.ok("Order created successfully.");
    }
}
```

## 🔐 Bonus: Ensure Uniqueness at Database Level

To prevent race conditions:

```sql
CREATE UNIQUE INDEX idx_idempotency_key ON idempotency_record(idempotency_key);
```

## 🧪 Testing the Idempotency

- Send the same request twice with the same `Idempotency-Key` header.
- First call processes the request.
- Second call returns the cached/stored response or a message.

## 💡 Best Practices

- Expire idempotency keys after a defined time (e.g., 24 hours).
- Only make `POST`, `PUT`, or `PATCH` idempotent.
- Use UUID or a hash of the request body as the idempotency key.

➔ The **idempotency key** is **typically generated by the client (frontend/UI)** and sent with each request to the backend.

## 🤷‍♂️ Who Sends the Idempotency Key?

| Option | Who Generates? | Notes |
|---|---|---|
| ✅ **Client (UI / Mobile / API consumer)** | Generates and sends a unique key per action | Most common and scalable |
| 🔁 Backend | Accepts request without key and generates one internally | Rare — not useful for retries since UI doesn't know the key |

## 🧠 Why Should the Client Send It?

1. **To support retries** safely (e.g., on timeouts or failures).
2. Backend **can't detect retries** if the request looks new each time.
3. Backend can **trust and verify** duplicate keys if client controls them.

## 🧪 Example: UI Sending Key

### JavaScript / Axios Example

```js
const idempotencyKey = crypto.randomUUID(); // Or use UUID library

axios.post('/orders', orderData, {
  headers: {
    'Idempotency-Key': idempotencyKey
  }
});
```

## 🧩 Alternative: Backend Generates Key and Returns It

If your use case requires it:

1. Client sends a request without the key.
2. Backend generates a key, stores it, returns it.
3. Client uses that key for retries.

But this is **less common** and harder to manage in distributed systems.

**44) Discuss Spring Boot's approach to database migrations. How can one integrate tools like Flyway or Liquibase?**

➔ Spring Boot integrates seamlessly with database migration tools like Flyway and Liquibase, automating the process of database schema creation and updates.

➔ To use Flyway, add its starter dependency to your pom.xml or build.gradle.
➔ Spring Boot automatically detects Flyway migrations in **src/main/resources/db/migration** and applies them accordingly.
➔ Spring Boot automatically detects Flyway migrations in **src/main/resources/db-changelog.sql** and applies them accordingly.

**45) How do you implement caching in Spring Boot for improved performance?**

## 🧠 What is Caching?

**Caching** is storing data **temporarily** to serve future requests **faster** without hitting the database or a slow service every time.

## 🧰 Why Use Caching?

- Reduce database or API calls
- Improve response times
- Handle high traffic efficiently
- Save computational cost

🚀 Levels of Caching

| Level | Example | Description |
|---|---|---|
| In-memory | `@Cacheable`, `ConcurrentHashMap` | Local to the app |
| Distributed | Redis, Hazelcast | Shared across microservices |
| Persistent | CDN, HTTP cache headers | For static assets |

## ✅ Step-by-Step: Basic Caching in Spring Boot

### 🔹 Step 1: Add Dependency

xml

```xml
<!-- Add in your pom.xml -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-cache</artifactId>
</dependency>
```

For Redis (later):

xml

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

### 🔹 Step 2: Enable Caching

java

```java
@SpringBootApplication
@EnableCaching
public class MyApp {
    public static void main(String[] args) {
        SpringApplication.run(MyApp.class, args);
    }
}
```

### 🔹 Step 3: Use `@Cacheable`

java

```java
@Service
public class ProductService {

    @Cacheable(value = "products", key = "#id")
    public Product getProductById(Long id) {
        simulateSlowService(); // e.g., DB or external call
        return productRepository.findById(id).orElse(null);
    }

    private void simulateSlowService() {
        try { Thread.sleep(3000); } catch (InterruptedException e) {}
    }
}
```

### ➕ Explanation

- `@Cacheable`: Checks the cache before executing the method.
- `value`: Cache name.
- `key`: Key for the cache entry (`#id` = method parameter).

## Other Cache Annotations

| Annotation | Use-case |
| --- | --- |
| `@Cacheable` | Read from cache or method |
| `@CachePut` | Always run method & update cache |
| `@CacheEvict` | Remove from cache |

```
@CacheEvict(value = "products", key = "#id")
public void deleteProduct(Long id) {
    productRepository.deleteById(id);
}
```

## 46) What are the best practices for managing application properties and secrets in Spring Boot?

➔ Managing application properties and secrets in Spring Boot is crucial for security, maintainability, and environment-specific configuration.

### 🧩 1. Organize application.properties or application.yml

Split configurations by environment:

```
arduino

src/
├── main/
│   └── resources/
│       ├── application.yml          # default
│       ├── application-dev.yml      # dev-specific
│       └── application-prod.yml     # prod-specific
```

### 🔐 2. Never Hardcode Secrets

Bad practice:

```yaml
db:
  username: admin
  password: mySecret123
```

### ✅ 3. Use Environment Variables

Let external systems pass secrets:

```yaml
spring:
  datasource:
    username: ${DB_USER}
    password: ${DB_PASS}
```

Then set in your environment:

```bash
export DB_USER=admin
export DB_PASS=superSecret
```

### 🔄 4. Use `@ConfigurationProperties` for Structured Config

Encapsulate grouped settings:

```java
@ConfigurationProperties(prefix = "payment")
@Component
public class PaymentProperties {
    private String apiKey;
    private String merchantId;
}
```

```yaml
payment:
  apiKey: ${PAYMENT_API_KEY}
  merchantId: 12345
```

## 🔐 5. **Use a Secret Manager for Production**

Use services like:

- ✅ AWS Secrets Manager
- ✅ Azure Key Vault
- ✅ HashiCorp Vault
- ✅ Spring Cloud Config with encryption

📦 Example: **Spring Cloud Vault**

```yaml
spring:
  cloud:
    vault:
      uri: http://localhost:8200
      token: ${VAULT_TOKEN}
```

## 📁 6. **Secure `application.yml` Files**

- Never commit secrets to Git
- Add `application-prod.yml` to `.gitignore` if it contains sensitive data
- Use `.env` files only for **local dev**, not for production

## 📄 7. **Use Profiles with `@Profile` Annotation**

Activate beans per environment:

```java
@Profile("dev")
@Bean
public DataSource devDataSource() { ... }

@Profile("prod")
@Bean
public DataSource prodDataSource() { ... }
```

## ✅ Summary: Best Practices

| Practice | Benefit |
| --- | --- |
| Use profiles ( `application-dev.yml` ) | Clean separation per environment |
| Store secrets in env vars or Vault | Secure, scalable, CI/CD friendly |
| Use `@ConfigurationProperties` | Organized, type-safe configs |
| Validate properties | Avoid runtime config errors |
| Use encryption for sensitive data | Security for in-code secrets |
| Avoid hardcoding or committing secrets | Prevent leaks |

## 47 ) What are the tools available for detecting memory leak ?

### VisualVM

➔ VisualVM is an all-in-one Java troubleshooting tool that integrates several JDK command-line tools .
➔ It is lightweight performance and memory profiling capabilities.
➔ It's included in the Oracle JDK download.

**Key Features:**

- Monitor application memory consumption in real-time.
- Analyze heap dumps to identify memory leaks.
- Track down memory leaks with its built-in heap walker.

### Eclipse Memory Analyzer (MAT)

➔ Eclipse MAT is a specialized tool designed for analyzing heap dumps.
➔ It is particularly effective in identifying memory leaks and reducing memory consumption.

**Key Features:**

- Analyze large heap dumps.
- Automatically identify memory leak suspects.
- Provide detailed reports on memory consumption by objects.

**Usage Example:**

**After obtaining a heap dump from a running application**

**(which can be triggered in the JVM on OutOfMemoryError),**

**MAT can be used to analyze this dump.**

**It provides a histogram of objects in memory, allowing developers to see which classes and objects are consuming the most memory.**

**JProfiler**

**➔ JProfiler is a comprehensive profiling tool for Java with capabilities for both memory and performance profiling.**
**➔ It's a commercial tool but is widely regarded for its user-friendly interface and detailed analysis.**

**Key Features:**

- **Real-time memory and CPU profiling.**
- **Advanced heap analysis and visualization.**
- **Ability to track every object in the heap and analyze memory consumption.**

## 48 ) How do you implement a custom health check in Spring Boot Actuator?

➔ To implement a custom health check in Spring Boot Actuator, you need to create a bean that implements the **HealthIndicator** interface.

➔ Spring Boot will automatically include your custom health check in the /actuator/health endpoint.

```java
@Component
public class MyCustomHealthIndicator implements HealthIndicator {

    @Override
    public Health health() {
        // Perform some custom logic (e.g., DB check, external service ping)
        boolean isServiceUp = checkCustomServiceHealth();

        if (isServiceUp) {
            return Health.up()
                         .withDetail("customService", "Available")
                         .build();
        } else {
            return Health.down()
                         .withDetail("customService", "Not reachable")
                         .build();
        }
    }

    private boolean checkCustomServiceHealth() {
        // Replace with real check
        return true;
    }
}
```

### ⚙️ 3. Enable Health Details in `application.yml`

yaml

```yaml
management:
  endpoints:
    web:
      exposure:
        include: health,info
  endpoint:
    health:
      show-details: always
```

**49) How would you handle a situation where a Spring Boot application runs out of memory?**

→ When a Spring Boot application runs out of memory (OOM), it's often due to memory leaks, inefficient data structures, or misconfigured JVM settings.

## 🛠️ Step 1: **Identify the Root Cause**

### 🔍 Check the error logs:

Look for `java.lang.OutOfMemoryError` in logs:

- `OutOfMemoryError: Java heap space`
- `OutOfMemoryError: GC overhead limit exceeded`
- `OutOfMemoryError: Metaspace`

### 📈 Enable heap dump on OOM:

Add to JVM options:

```
bash

-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/path/to/dumps
```

Analyze heap dump with:

- Eclipse MAT (Memory Analyzer Tool)
- VisualVM
- JProfiler

## 🧪 Step 2: **Analyze Memory Usage**

Use runtime tools:

- `jstat`, `jmap`, `jstack`, `jconsole`
- Spring Boot Actuator: enable `/actuator/heapdump`, `/metrics`

## ⚙️ Step 3: **Configure Appropriate JVM Settings**

Set appropriate heap size in `application startup` or Docker config:

```
bash

-Xms512m -Xmx2g
```

Also consider:

```
bash

-XX:MaxMetaspaceSize=256m
```

## 🔁 Step 4: **Code-Level Fixes**

### 🚫 1. Avoid loading large datasets into memory:

```java
// ❌ Avoid
List<Product> products = productRepository.findAll();

// ✅ Use pagination or streaming
Page<Product> products = productRepository.findAll(PageRequest.of(0, 100));
```

## 50) How would you design a Spring Boot application to handle millions of concurrent users?

→ Designing a Spring Boot application to handle millions of concurrent users requires careful planning across architecture, scalability, performance optimization, and fault tolerance.

### 🧱 1. **Use a Microservices or Modular Monolith Architecture**

Break the application into functional services:

- User Service
- Product/Catalog Service
- Order Service
- Payment Service
- Inventory Service
- Notification Service

**Benefits:**

- Scalability per feature
- Fault isolation
- Easier deployments

### 🧱 2. **API Gateway and Load Balancer**

- Use **Spring Cloud Gateway** or **Kong/Nginx** as the API Gateway
- Use **AWS ELB, Azure Application Gateway**, or GCP Load Balancing for traffic distributior

Key Features:

- Rate limiting
- IP filtering
- Authentication/authorization
- Routing and versioning

### ⚙️ 3. **Scalability Strategy**

Horizontal Scaling:

- Use **containers (Docker)** and **orchestration (Kubernetes)**
- Autoscale with **Horizontal Pod Autoscaler (HPA)** based on CPU/memory/requests

Stateless Services:

- Design services to be **stateless** so they can scale easily
- Store session state in **Redis** or another distributed cache

## 🧠 4. **Database Design for High Throughput**

- Use **sharded** or **partitioned** databases
- Read/write separation (replication)
- Connection pooling (HikariCP)
- Use **NoSQL (e.g. Cassandra, DynamoDB)** for high-velocity data

**Optimize:**

- Indexes
- Query execution plans
- Avoid joins and heavy aggregations in high-throughput paths

## 🚀 5. **Use Caching Aggressively**

Reduce DB hits with:

- `@Cacheable` + **Caffeine/Redis**
- CDN for static content (CloudFront, Akamai)

Example:

```java
@Cacheable(value = "products", key = "#id")
public Product getProductById(Long id) { ... }
```

## 🔁 6. **Asynchronous Processing**

Offload non-critical workloads using:

- **Kafka**, **RabbitMQ**, or **AWS SQS**
- Use Spring Boot with `@Async`, `@Scheduled` for background tasks

Example use cases:

- Email notifications
- Order processing
- Event logging

## 🔐 7. **Security and Rate Limiting**

- Use **JWT + OAuth2** for authentication
- Use **API Gateway** or filters for throttling
- Protect endpoints with **Spring Security** and custom filters

## 🔁 9. **Resilience and Failover**

- Use **Resilience4j** for:
  - Circuit breakers
  - Rate limiters
  - Retry logic
- Use **K8s readiness/liveness probes**
- Geo-redundant deployment (multi-AZ, multi-region)
- Graceful fallback mechanisms

## 🧪 10. **Performance Tuning**

- Tune **JVM parameters** ( `-Xmx`, GC, etc.)
- Use **non-blocking WebClient** (instead of RestTemplate)
- Tune thread pools and database connection pools
- Compress responses ( `gzip` )
- Batch expensive operations

## 51) What are the best practices for logging in Spring Boot?

### 1. Use a Standard Logging Framework

- Spring Boot uses **SLF4J** as the logging facade by default.
- Common implementations:
  - **Logback** (default)
  - Log4j2 (optional)
- Stick to **SLF4J API** in your code for flexibility.

```java
private static final Logger logger = LoggerFactory.getLogger(MyClass.class)
logger.info("This is a log message");
```

### 2. Configure Logging Levels Appropriately

- Use levels **TRACE, DEBUG, INFO, WARN, ERROR** judiciously.
- Avoid DEBUG/TRACE in production unless troubleshooting.
- Configure levels in `application.properties` or `application.yml`.

```properties
logging.level.org.springframework.web=DEBUG
logging.level.com.myapp=INFO
```

### 3. Externalize Log Configuration

- Use `logback-spring.xml` or `log4j2-spring.xml` for advanced config.
- Avoid hardcoding settings in code.
- Enable easy changes without redeploying.

### 8. Log Exceptions Properly

- Always log the exception stack trace, not just the message.

```java
try {
  // risky code
} catch (Exception e) {
  logger.error("Error processing request", e);
}
```

## 52) What are the best practices for securing a Spring Boot application?

→ Securing a Spring Boot application involves multiple layers—from authentication and authorization to protecting against common vulnerabilities and securing infrastructure.

### 1. Use Spring Security

- Leverage **Spring Security**, the de-facto security framework for Spring apps.
- Configure authentication and authorization with roles and permissions.
- Use **OAuth2 / OpenID Connect** with providers like Keycloak, Okta, or Auth0 for SSO.
- Prefer **stateless JWT tokens** for REST APIs.

### 2. Secure Authentication

- Store passwords securely using **bcrypt** or other strong hashing algorithms.
- Implement **multi-factor authentication** (MFA) if needed.
- Limit login attempts to prevent brute force attacks.

### 3. Secure Authorization

- Use **method-level security** ( `@PreAuthorize` , `@Secured` ) to restrict sensitive operations.
- Enforce **principle of least privilege**: users get only the permissions they need.
- Validate permissions both on frontend and backend.

### 4. Protect Against Common Vulnerabilities

- **SQL Injection**: Use **parameterized queries** or Spring Data repositories instead of string concatenation.
- **Cross-Site Scripting (XSS)**: Sanitize inputs, use frameworks' escaping features, enable Content Security Policy (CSP).
- **Cross-Site Request Forgery (CSRF)**: Enable CSRF protection on web forms (Spring Security enables it by default).
- **Clickjacking**: Use HTTP headers like `X-Frame-Options` or Spring Security's defaults.
- **Secure Headers**: Set HTTP security headers ( `Strict-Transport-Security` , `X-Content-Type-Options` , etc.)

### 5. Use HTTPS

- Always serve your app over **HTTPS** with valid certificates.
- Redirect HTTP traffic to HTTPS.

## 53) How do you ensure code quality in a Spring Boot project?

### 1. **Follow Clean Code Principles**

- Use meaningful variable and method names.
- Keep methods small and focused on a single responsibility.
- Avoid duplicate code (DRY principle).
- Structure code into logical layers: Controller, Service, Repository.

### 2. **Use Static Code Analysis Tools**

- Integrate tools like:
  - **SonarQube** – for code smells, bugs, and coverage.
  - **Checkstyle** – for enforcing coding standards.
  - **PMD/FindBugs (SpotBugs)** – for detecting bugs and anti-patterns.
- Run analysis in CI pipelines for automatic feedback.

### 3. **Write Unit Tests**

- Use **JUnit 5** and **Mockito** to test services and utility classes.
- Aim for high coverage, but focus on meaningful tests.

### 4. **Write Integration Tests**

- Use **Spring Boot Test** ( `@SpringBootTest` , `@WebMvcTest` ) to test APIs and database interactions.
- Run tests against in-memory databases (e.g., H2) for speed.

### 5. **Use Code Reviews**

- Enforce pull requests (PRs) with mandatory code reviews.
- Review code for:
  - Logic correctness
  - Code clarity
  - Performance
  - Security
  - Test coverage

### 6. **Adopt CI/CD with Quality Gates**

- Use CI tools (e.g., GitHub Actions, Jenkins, GitLab CI) to:
  - Run tests
  - Lint code
  - Run static analysis
- Use SonarQube quality gates to block merging code with issues.

### 7. **Use Consistent Code Formatting**

- Use IDE formatters or tools like **Spotless** or **Prettier** for Java.
- Enforce formatting using pre-commit hooks or CI pipelines.

### 8. **Document the Code**

- Use Javadoc for public APIs.
- Document assumptions, edge cases, and important business logic.

### 10. **Monitor Code Metrics**

Track:

- Code coverage
- Code duplication
- Cyclomatic complexity
- Number of issues over time

## 55) How do you implement OAuth2 authentication in Spring Boot?

### ✅ Steps to Implement OAuth2 Client Authentication in Spring Boot

#### 🔐 OAuth2 Scenarios

You can configure Spring Boot to:

1. Authenticate users via an **OAuth2 provider** (Google, GitHub, Keycloak).
2. Secure your REST APIs using **JWT tokens** (resource server).
3. Act as an **authorization server** (with Spring Authorization Server).

#### 1. Add Dependencies

```xml
<!-- OAuth2 Client -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-oauth2-client</artifactId
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

#### 2. Configure `application.yml` or `application.properties`

Example: Login with Google

```yaml
spring:
  security:
    oauth2:
      client:
        registration:
          google:
            client-id: YOUR_GOOGLE_CLIENT_ID
            client-secret: YOUR_GOOGLE_CLIENT_SECRET
            scope: profile, email
        provider:
          google:
            authorization-uri: https://accounts.google.com/o/oauth2/v2/auth
            token-uri: https://oauth2.googleapis.com/token
            user-info-uri: https://www.googleapis.com/oauth2/v3/userinfo
```

#### 3. Enable OAuth2 Login

No extra configuration is required. Spring Boot auto-configures a login page at `/login`.

### 4. Protect Routes (Optional Custom Config)

java Copy

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
      .authorizeHttpRequests(auth -> auth
          .requestMatchers("/", "/public").permitAll()
          .anyRequest().authenticated()
      )
      .oauth2Login();  // Enable OAuth2 login
    return http.build();
  }
}
```

## 5. Access Authenticated User Details

java

```java
@GetMapping("/profile")
public String userDetails(@AuthenticationPrincipal OAuth2User principal) {
    return "Hello " + principal.getAttribute("name");
}
```

## ✅ Securing REST APIs (OAuth2 Resource Server)

### 1. Add Dependency

xml

```xml
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
```

### 2. Configure JWT Issuer (e.g., Keycloak, Auth0, Okta)

yaml

```yaml
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: https://your-auth-domain/.well-known/openid-configuratio
```

### 3. Secure Endpoints

java

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Ex
    http
      .authorizeHttpRequests(auth -> auth
          .anyRequest().authenticated()
      )
      .oauth2ResourceServer()
      .jwt();
    return http.build();
  }
}
```

### Optional: Role-Based Access Control

Use JWT claim mappings or Keycloak roles:

java

```java
@PreAuthorize("hasRole('ADMIN')")
@GetMapping("/admin")
public String adminOnly() {
    return "Admin content";
}
```

## 56 ) What is the difference between Basic Authentication and JWT in Spring Boot?

➔ The difference between Basic Authentication and JWT (JSON Web Token) in Spring Boot lies in how authentication information is transmitted, stored, and managed.

### 🔐 1. Basic Authentication

#### ✅ How It Works

- Client sends username and password in the `Authorization` header of every HTTP request:

```
pgsql                                                    Copy

Authorization: Basic base64(username:password)
```

- Server validates credentials on **every request**.

#### ⚠️ Risks

- Less secure, especially over unsecured (non-HTTPS) connections.
- Can't easily revoke access without changing password.

### ✅ Characteristics

| Feature | Description |
|---|---|
| State | Stateless (but credentials sent every time) |
| Credential Reuse | Password is sent with every request (security risk |
| Token Format | No token; base64-encoded credentials |
| Expiration | None (unless managed externally) |
| Server Storage | Not needed |
| Use Case | Simple internal services, basic auth testing |

## 🔐 2. JWT (JSON Web Token) Authentication

### ✅ How It Works

- User authenticates once → server issues a **JWT** (digitally signed token).
- Client sends the token in the `Authorization` header:

```makefile
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI...
```

- Server verifies the token's signature and claims on each request.

### ✅ Characteristics

| Feature | Description |
|---|---|
| State | Fully stateless |
| Credential Reuse | Token is reused (not password) |
| Token Format | Self-contained (contains user info, roles, expiry) |
| Expiration | Yes (usually short-lived) |
| Server Storage | No need (unless using blacklists) |
| Use Case | Scalable APIs, mobile/backend auth, microservices |

### 🔍 Feature Comparison

| Feature | Basic Authentication | JWT Authentication |
|---|---|---|
| Auth Header | `Authorization: Basic` | `Authorization: Bearer` |
| Credential Sent | Every request (username/password) | Once (token issued) |
| Stateless | Yes | Yes |
| Security Risk | High (password reuse) | Lower (no password in each reque |
| Revocation | Change password | Harder; needs token blacklist/cach |
| Use with OAuth2 | Not compatible | Standard approach |
| Token Content | None | Claims like user ID, roles, expiry |
| Best Use | Dev, test, simple internal apps | Production APIs, scalable services |

### 🛠️ Spring Boot Implementation

- **Basic Auth**: Use `HttpBasic` in security config

```java
http
  .authorizeHttpRequests().anyRequest().authenticated()
  .and().httpBasic();
```

- **JWT**: Use `spring-boot-starter-oauth2-resource-server`

```yaml
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: https://example.com/auth
```

## 57) How do you implement role-based access control (RBAC) in Spring Boot?

## 58) How do you optimize database queries in Spring Boot using JPA or Hibernate?

### ✅ 1. Use Projections Instead of Fetching Entire Entitie

Fetch only the required fields using **DTO projections**.

❌ Bad:

```java
List<User> users = userRepository.findAll();  // Fetches entire entity
```

✅ Optimized:

```java
public interface UserNameEmailProjection {
    String getName();
    String getEmail();
}
List<UserNameEmailProjection> findByActiveTrue();
```

### ✅ 2. Use Pagination for Large Result Sets

Avoid returning thousands of rows at once.

```java
Page<User> findByStatus(String status, Pageable pageable);
```

```java
Pageable pageable = PageRequest.of(0, 20);
Page<User> users = userRepository.findByStatus("ACTIVE", pageable);
```

## ✅ 3. **Avoid N+1 Problem Using Fetch Joins**

N+1 happens when you fetch a list of entities and each entity triggers additional queries for related entities.

### ❌ Bad:

java Copy Edit

```java
List<Order> orders = orderRepository.findAll();  // Then multiple SELECTs for each order items
```

### ✅ Optimized with Join Fetch:

java Copy Edit

```java
@Query("SELECT o FROM Order o JOIN FETCH o.items")
List<Order> findAllWithItems();
```

Or annotate entity with:

java

```java
@OneToMany(fetch = FetchType.LAZY)
@JoinFetch
private List<Item> items;
```

## ✅ 7. **Batch Inserts/Updates**

educe number of insert/update queries:

yaml

```yaml
spring.jpa.properties.hibernate.jdbc.batch_size=30
spring.jpa.properties.hibernate.order_inserts=true
spring.jpa.properties.hibernate.order_updates=true
```

ntity:

java

```java
@BatchSize(size = 30)
```

## ✅ 8. **Use Native Queries for Complex Joins**

If JPA queries are too slow or unreadable:

java

```java
@Query(value = "SELECT * FROM users WHERE status = 'ACTIVE'", nativeQuery = true)
List<User> findActiveUsers();
```

## ✅ 10. **Cache Frequently Accessed Data**

Use 2nd level cache or Spring Cache abstraction:

```java
@Cacheable("users")
public User getUserById(Long id) { … }
```

## ✅ Summary of Optimization Techniques

| Technique | Benefit |
| --- | --- |
| DTO/Projections | Reduces memory and data transfer |
| Pagination | Handles large datasets efficiently |
| Fetch Joins & `@EntityGraph` | Avoids N+1 queries |
| SQL Logging | Helps identify slow or inefficient SQL |
| Batch Operations | Reduces DB round trips |
| Indexes | Speeds up queries |
| Native Queries | More control and performance |
| Caching | Reduces DB load for frequently accessed data |

**58) How would you handle a situation where a Spring Boot application fails to start due to a circular dependency?**

## How to Handle Circular Dependency in Spring Boot

### 1. Refactor Your Code to Remove Circular Dependencies

- **Best practice:** redesign your classes so dependencies flow in one direction.
- Use interfaces and abstractions to break tight coupling.
- For example, introduce a third bean that both depend on instead of depending on each other.

### 3. Use Setter Injection Instead of Constructor Injection

Setter injection allows Spring to create beans first and then inject dependencie

java

```
@Component
public class BeanA {
    private BeanB beanB;

    @Autowired
    public void setBeanB(BeanB beanB) {
        this.beanB = beanB;
    }
}

@Component
public class BeanB {
    private BeanA beanA;

    @Autowired
    public void setBeanA(BeanA beanA) {
```

### 2. Use `@Lazy` Annotation

Mark one or both dependencies as lazy to delay injection until needed, allowing Spring to create pr
break the cycle.

java Copy

```
@Component
public class BeanA {
    private final BeanB beanB;

    public BeanA(@Lazy BeanB beanB) {
        this.beanB = beanB;
    }
}

@Component
public class BeanB {
    private final BeanA beanA;

    public BeanB(@Lazy BeanA beanA) {
        this.beanA = beanA;
    }
```