

# Springboot microservices guide

Below is the **FULL EXPANDED SET B (Q26–50)** with long, detailed, interview-ready answers.

Once Sets C & D are done, I will generate the **complete printable PDF book** (all 100 long answers).

---

---

## ✓ SET B — Spring Boot Microservices Implementation (Q26–50)

---

### 26. Explain the internal architecture of Spring Boot.

Spring Boot simplifies Spring development using:

#### 1. Auto-Configuration

One of its biggest advantages.

It scans classpath dependencies (e.g., spring-web, spring-data-jpa) and automatically configures:

- DispatcherServlet
- JPA EntityManager
- DataSource
- Jackson JSON converters
- Error handling

Reduces boilerplate drastically.

#### 2. Spring Boot Starters

Bundles of commonly-used dependencies:

- `spring-boot-starter-web`
- `spring-boot-starter-actuator`
- `spring-boot-starter-data-jpa`

- `spring-boot-starter-security`

A starter brings **consistent versions** and eliminates dependency conflicts.

### 3. Embedded Servers

Spring Boot runs inside:

- Tomcat (default)
- Jetty
- Undertow

No external server deployment → packaged as fat JAR.

### 4. Actuator

Production-ready endpoints:

- `/actuator/health`
- `/actuator/metrics`
- `/actuator/loggers`
- `/actuator/httptrace`
- `/actuator/prometheus`

Crucial for microservice observability.

### 5. Externalized Configuration

Config files from:

- `application.properties`
- `application.yml`
- environment variables
- Spring Cloud Config
- Vault

Priority-based resolution → consistent cross-environment configuration.

---

## 27. How does Spring Boot simplify building REST APIs?

Spring Boot + Spring MVC gives:

### 1. Annotation-based controllers

```
java
```

```
@RestController
@RequestMapping("/products")
public class ProductController {}
```

## 2. Built-in JSON serialization

Powered by Jackson.

Automatically converts Objects ↔ JSON.

## 3. Request validation

Using:

- `@Valid`
- Hibernate Validator

## 4. Exception handling

`@ControllerAdvice` for global error formats.

## 5. Request & Response objects

DTO mappings reduce exposure of internal entities.

## 6. Auto-configured Web Server

No need to deploy WAR — just run the JAR.

All these make REST development extremely fast and maintainable.

---

# 28. What is Spring Cloud? Why is it used in microservices?

Spring Cloud is a suite of tools that address challenges of distributed systems:

Spring Cloud Component	Purpose
Eureka	Service discovery
Config Server	Centralized configuration

Spring Cloud Component	Purpose
Gateway	Routing, auth, rate limiting
LoadBalancer	Client-side load balancing
Resilience4j	Circuit breakers, retries
Sleuth + Zipkin	Distributed tracing
Spring Cloud Bus	Refresh events, dynamic config
Feign	HTTP client with load balancing

Spring Cloud = enterprise-grade microservices toolkit.

## 29. Difference between RestTemplate, WebClient, and Feign?

Feature	RestTemplate	WebClient	Feign
Type	Blocking	Non-blocking	Declarative
Thread Model	Servlet	Reactive	Servlet/Reactive
Difficulty	Medium	High	Easiest
Load Balancing	Deprecated Ribbon	Manual	Built-in
Best Use	Simple sync calls	High concurrency	Microservices communication

**RESTTemplate is deprecated → prefer WebClient or Feign.**

## 30. How do you build resilient microservices with Resilience4j?

Resilience4j offers:

- Circuit breaker
- Rate limiter
- Retry
- Time limiter
- Bulkhead

### Example (Circuit Breaker):

java

```
@CircuitBreaker(name = "paymentService", fallbackMethod = "fallback")
public PaymentResponse callPayment() {...}
```

### Fallback:

java

```
public PaymentResponse fallback(Throwable t) {
    return new PaymentResponse("Payment temporarily unavailable");
}
```

### Why Needed:

- Avoid cascading failures
- Auto recovery
- Handle remote-service slowdowns

---

## 31. What are Feign Clients? Explain with real use case.

Feign is a declarative HTTP client.

### Example:

java

```
@FeignClient(name = "inventory-service")
public interface InventoryClient {
```

```
@GetMapping("/inventory/{id}")
ProductInventory getStock(@PathVariable Long id);
}
```

### Why used:

- Simplifies inter-service communication
- No manual WebClient code
- Integrated with:
  - Load balancing
  - Circuit breakers
  - Security interceptors

### Real E-Commerce use case:

Order-service → calls Inventory-service to check stock using Feign.

---

## 32. How do you externalize configuration in Spring Boot?

Priority order:

1. Command line arguments
2. Environment variables
3. application-{profile}.yml
4. application.yml
5. Default properties in JAR

### With Spring Cloud Config

bootstrap.yml :

```
yaml

spring:
  cloud:
    config:
      uri: http://config-server
```

Supports:

- Centralization

- Dynamic refresh ( /actuator/refresh )
  - Secret encryption
- 

## 33. How do you handle exceptions globally in microservices?

Use `@ControllerAdvice` .

### Example:

```
java

@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<?> handleNotFound(ResourceNotFoundException ex) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND)
            .body(new ErrorResponse(ex.getMessage(), "404"));
    }
}
```

### Benefits:

- Consistent error output
  - Reduced boilerplate
  - Clear API documentation
- 

## 34. How do you validate incoming requests?

Use:

- `@Valid`
- `@NotNull` , `@Size` , `@Email` , `@Min` , etc.

### Example:

```
java

public class UserRequest {
    @NotNull
```

```
@Size(min = 3)
private String name;

@email
private String email;
}
```

If validation fails → Spring throws `MethodArgumentNotValidException`.

Handled globally via `@ControllerAdvice`.

---

## 35. What are interceptors and filters? Use cases?

### Filters (Servlet level)

- Authenticate request
- Logging
- CORS handling
- Runs *before controller mapping*

### Interceptors (Spring MVC level)

- Modify request/response
- Add headers (Correlation-ID)
- Logging entry/exit
- Runs *after handler mapping*

### Flow:

nginx

Filters → Interceptors → Controller → Interceptors → Filters

---

## 36. Explain Spring Boot Actuator in depth.

Actuator exposes operational endpoints:

### Health

`/actuator/health`

- DB health
- Redis health
- Kafka health

## Metrics

`/actuator/metrics`

- JVM
- GC
- HTTP requests
- Database pool usage

## Prometheus

`/actuator/prometheus` integrates with Prometheus.

## Env

Shows all active environment variables.

## Loggers

Dynamically change log level:

```
bash
```

```
POST /actuator/loggers/com.example
{ "configuredLevel": "DEBUG" }
```

Actuator = foundation of microservice monitoring.

---

## 37. How do you secure a Spring Boot REST API using JWT?

### Steps:

1. Client sends credentials
2. Server generates JWT
3. Client stores token
4. Sends token in `Authorization: Bearer <JWT>`
5. Filter validates JWT on every request

## Configure filter:

```
java

public class JwtFilter extends OncePerRequestFilter {
    protected void doFilterInternal(...) {
        // validate token, set auth
    }
}
```

### Benefits:

- No session required
- Stateless
- Fast authentication

Used heavily in microservices.

---

## 38. What is Rate Limiting? How to implement it?

Prevent excessive API calls.

### Approaches:

1. API Gateway Rate Limiting
2. Redis Rate Limiter
3. Bucket4j in Spring Boot

### Example with Bucket4j:

```
java

@Bean
public Filter bucketFilter() {
    return new BucketFilter(Bandwidth.simple(10, Duration.ofMinutes(1)));
}
```

### Use cases:

- Prevent DDOS
  - Protect expensive endpoints
  - Enforce fair usage
-

## 39. How do you perform API versioning? Best practices?

### Methods:

#### 1. URI-based

```
bash

/v1/products
/v2/products
```

#### 2. Header-based

```
makefile

Accept-Version: 2
```

#### 3. Query param

```
bash

/products?version=2
```

### Best Practices:

- Keep old versions stable
- Document deprecation timeline
- Avoid breaking clients
- Keep versions simple (v1, v2)

---

## 40. Why should microservices NOT share a database?

Because it leads to:

- Tight coupling
- Hard deployments
- Cross-service schema changes
- Deadlocks across services
- No independent scaling

Each service should own:

- Its DB

- Its migrations

Communicate via **events**, not shared tables.

---

## 41. How to design pagination and filtering in REST APIs?

### Example:

```
bash
```

```
GET /products?page=0&size=10&sort=price,asc&category=electronics
```

Response includes:

- items
- page info
- total pages

### With Spring Data:

```
java
```

```
Page<Product> findByCategory(String cat, Pageable pageable);
```

---

## 42. What is the Outbox Pattern? Why used?

Prevents event loss during DB commit.

### Problem:

Service writes DB record then publishes event  
→ but event may fail → inconsistent system.

### Solution:

1. Write record + event to same DB transaction (outbox table)
2. Event Relay polls outbox table
3. Publishes events to Kafka
4. Marks as sent

**Benefits:**

- 100% reliable events
  - No dual-write issues
  - Ensures eventual consistency
- 

## 43. How to connect microservices using asynchronous communication?

**Tools:**

- Kafka
- RabbitMQ
- AWS SQS
- Google Pub/Sub

**Patterns:**

- Event-driven architecture
- CQRS
- Saga

**Benefits:**

- Loose coupling
  - Better resilience
  - Zero downtime scaling
- 

## 44. Explain logging best practices in microservices.

**Use JSON logs:**

json

```
{"timestamp": "...", "traceId": "...", "service": "order-service", "level": "INFO", "msg": "Order created"}
```

**Always add:**

- correlation-id
- trace-id

- service name
- request path
- response status
- execution time

### Centralized logging tools:

- ELK (Elasticsearch + Logstash + Kibana)
  - Loki + Grafana
  - Splunk
- 

## 45. How do you implement a custom exception response format?

### Standard Response:

```
json

{
  "timestamp": "...",
  "error": "VALIDATION_FAILED",
  "message": "Email is invalid",
  "path": "/users"
}
```

### Implemented through:

- `@ControllerAdvice`
  - Custom error object
- 

## 46. How do you test microservices? (Unit, Integration, Contract)

### Unit Testing

- Mockito + JUnit
- Test only business logic

### Integration Testing

- `@SpringBootTest`

- Embedded Mongo/Redis
- Test full request flow

## Contract Testing

Use **Spring Cloud Contract** to ensure API compatibility.

## Component Testing

Test each microservice independently with stubbed dependencies.

---

# 47. How do you build asynchronous REST APIs?

Methods:

1. `DeferredResult`
2. `CompletableFuture`
3. `WebFlux Mono/Flux`
4. Kafka-driven async workflows

## Example:

```
java

@Async
public CompletableFuture<OrderResponse> placeOrder() { ... }
```

Used in:

- Long-running processes
  - Heavy I/O operations
- 

# 48. What is Spring Boot DevTools?

Makes development faster:

- Auto reload on changes
- LiveReload browser refresh
- Faster restart time

Not used in production.

---

## 49. How to improve performance of Spring Boot microservices?

### Techniques:

- Use connection pooling (HikariCP)
  - Use WebClient instead of RestTemplate
  - Add caching (Redis)
  - Tune JVM (Xmx, GC)
  - Use async calls
  - Reduce serialization overhead (Jackson)
  - Enable compression
  - Use bulkheads
- 

## 50. What are Spring Profiles? How do you use them?

Profiles = environment-specific configs.

### Example:

```
application-dev.yml
application-prod.yml
```

Activate using:

- `--spring.profiles.active=dev`
- Environment variable
- In bootstrap.yml

### Use cases:

- Different DBs per environment
  - Feature toggles
  - Logging levels
- 



**SET B Completed (25 questions)**

Next: SET C (Q51–75: Messaging, Data, Scalability, Deployment)

Then: SET D (Q76–100: Observability, DevOps, Kubernetes, Cloud)

---

👉 Reply: “Continue Set C” to get the next long-answer set.