1️ What is Spring Boot, and how is it different from Spring Framework?

Answer:

Spring Boot is an extension of the **Spring Framework** that simplifies the development of Java applications by providing **auto-configuration**, **embedded servers**, and a convention-over-configuration approach.

Key Differences:

Feature	Spring Framework	Spring Boot
Configuration	Requires XML/Java-based config	Uses auto-configuration
Server Setup	Needs an external server (Tomcat, Jetty)	Comes with an embedded server
Dependency Management	Manual dependency handling	Uses **Spring Boot Starter** dependencies
Microservices Support	Requires setup	Built-in support for microservices

Example:

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

This single class is enough to start a Spring Boot application.

2️ What is `@SpringBootApplication`, and what does it do?

Answer:

`@SpringBootApplication` is a **composite annotation** that combines:

- **`@Configuration`** → Defines beans in Spring
- **`@EnableAutoConfiguration`** → Automatically configures beans based on dependencies
- **`@ComponentScan`** → Scans components (`@Controller`, `@Service`, `@Repository`)

Example:

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

```
    }
}
```

This annotation removes boilerplate configurations and speeds up application development.

---

3 What is Dependency Injection (DI) in Spring Boot?

Answer:

Dependency Injection (DI) is a design pattern where **Spring automatically injects dependencies** into components, avoiding manual instantiation.

Example:

```
@Component
class ServiceA {
    public void printMessage() {
        System.out.println("Hello from ServiceA!");
    }
}

@Component
class ServiceB {
    private final ServiceA serviceA;

    @Autowired  // Dependency Injection
    public ServiceB(ServiceA serviceA) {
        this.serviceA = serviceA;
    }

    public void execute() {
        serviceA.printMessage();
    }
}
```

Why use DI?

- **Removes tight coupling**
- **Enhances testability**
- **Improves maintainability**

---

4 What is the difference between `@Component`, `@Service`, and `@Repository`?

Answer:

These are **stereotype annotations** that register classes as Spring Beans.

Annotation	Purpose
`@Component`	Generic annotation for Spring Beans. Used when no specific role is

	defined.
`@Service`	Used for business logic/service layer components.
`@Repository`	Used for DAO (Data Access Layer) components and enables exception translation.

Example:

```
@Component
class GeneralComponent {}

@Service
class MyService {}

@Repository
class MyRepository {}
```

These annotations help Spring **automatically manage beans** using `@ComponentScan`.

---

5 How does Spring Boot handle exception management in REST APIs?

Answer:

Spring Boot provides **global exception handling** using `@ControllerAdvice` and `@ExceptionHandler`.

Example:

```
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<String> handleNotFound(ResourceNotFoundException ex) {
        return new ResponseEntity<>(ex.getMessage(), HttpStatus.NOT_FOUND);
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<String> handleException(Exception ex) {
        return new ResponseEntity<>("Something went wrong",
HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

Why use it?

- Centralized error handling
- Custom error messages
- Consistent API responses

6️ How do you create a REST API in Spring Boot?

Answer:

Spring Boot makes REST API development simple using `@RestController` and `@RequestMapping`.

Example:

```
@RestController
@RequestMapping("/users")
public class UserController {

    @GetMapping("/{id}")
    public ResponseEntity<String> getUser(@PathVariable int id) {
        return ResponseEntity.ok("User with ID: " + id);
    }

    @PostMapping
    public ResponseEntity<String> createUser(@RequestBody String user) {
        return ResponseEntity.status(HttpStatus.CREATED).body("User Created: " +
user);
    }
}
```

Key Annotations:

- `@RestController` → Marks a class as a REST API controller
- `@GetMapping`, `@PostMapping`, etc. → Define HTTP methods
- `@PathVariable` → Extracts values from URL
- `@RequestBody` → Reads request body

7️ What is Spring Boot Actuator, and how is it used?

Answer:

Spring Boot **Actuator** provides built-in endpoints to monitor and manage applications.

Key Features:

- **Health Checks** (`/actuator/health`)
- **Metrics** (`/actuator/metrics`)
- **Environment Info** (`/actuator/env`)

How to Enable Actuator?

1️ Add the dependency:

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

2️ Configure `application.properties`:

```
management.endpoints.web.exposure.include=*
```

3️⃣ Access endpoints like:

```
http://localhost:8080/actuator/health
```

Actuator helps **monitor microservices** and improve observability.

8️⃣ What is Circuit Breaker in Microservices, and how do you implement it in Spring Boot?

Answer:

A **Circuit Breaker** prevents system failures by stopping calls to a failing service and providing a fallback response.

How to Implement Circuit Breaker using Resilience4j?

1️⃣ Add the dependency:

```
<dependency>
    <groupId>io.github.resilience4j</groupId>
    <artifactId>resilience4j-spring-boot2</artifactId>
</dependency>
```

2️⃣ Use `@CircuitBreaker` in your service:

```
@Service
public class InventoryService {

    @CircuitBreaker(name = "inventoryService", fallbackMethod = "fallbackInventory")
    public String checkInventory() {
        throw new RuntimeException("Service Down!");
    }

    public String fallbackInventory(Exception ex) {
        return "Fallback: Default Inventory Available";
    }
}
```

Why use Circuit Breaker?

✅ Prevents cascading failures
✅ Improves application **resilience**
✅ Ensures service **availability**

1️⃣ What are Microservices, and how are they different from Monolithic Architecture?

Answer:

Microservices architecture is a design approach where an application is divided into **small, independent services**, each performing a specific function. These services **communicate via APIs** and can be deployed, scaled, and updated independently.

Key Differences:

Feature	Monolithic Architecture	Microservices Architecture
Scalability	Difficult, scales as a whole	Easy, scale individual services
Deployment	Single large deployment	Independent deployments
Technology	Single tech stack	Can use multiple technologies
Fault Isolation	Failure affects the entire app	Failures are isolated to a single service
Development Speed	Slower due to dependencies	Faster, as teams work independently

Example:

- **Monolithic:** One large app handling users, payments, and orders.
- **Microservices:** Separate services for **users, payments, and orders**, communicating via REST or messaging.

---

2 What is an API Gateway in Microservices?

Answer:

An **API Gateway** is a single entry point that manages and routes client requests to the correct microservice.

Why use an API Gateway?

 Authentication & Authorization (e.g., JWT validation)
 Rate Limiting & Security (prevents excessive requests)
 Load Balancing (distributes requests efficiently)
 Logging & Monitoring (tracks request flow)

Example using Spring Cloud Gateway:

```
spring:
  cloud:
    gateway:
      routes:
        - id: user-service
          uri: lb://USER-SERVICE
          predicates:
            - Path=/users/**
```

This routes requests to **USER-SERVICE** when `/users/**` is called.

3️ What is Service Discovery, and how does Eureka work in Microservices?

Answer:

Service Discovery helps microservices find each other dynamically **without hardcoding URLs**. **Eureka Server** (Netflix Eureka) is a registry where services register themselves and discover other services.

How Eureka Works?

1️ **Eureka Server** acts as a registry.
2️ **Eureka Clients** (services) register themselves.
3️ Other services **query Eureka** to find service locations.

Example:

Eureka Server (`application.yml`)

```
server:
  port: 8761

eureka:
  client:
    register-with-eureka: false
    fetch-registry: false
```

Eureka Client (Microservice)

```
eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/
```

This allows services to register and discover each other dynamically.

4️ What is Circuit Breaker, and how do you implement it in Microservices?

Answer:

A **Circuit Breaker** prevents cascading failures by **stopping** calls to a failing service and returning a fallback response.

How to Implement Circuit Breaker using Resilience4j?

1️ Add the dependency:

```
<dependency>
    <groupId>io.github.resilience4j</groupId>
    <artifactId>resilience4j-spring-boot2</artifactId>
</dependency>
```

2️⃣ Use `@CircuitBreaker` in your service:

```
@Service
public class InventoryService {

    @CircuitBreaker(name = "inventoryService", fallbackMethod = "fallbackInventory")
    public String checkInventory() {
        throw new RuntimeException("Service Down!");
    }

    public String fallbackInventory(Exception ex) {
        return "Fallback: Default Inventory Available";
    }
}
```

Why use Circuit Breaker?

✅ **Prevents cascading failures**
✅ **Improves system resilience**
✅ **Ensures service availability**

5️⃣ How does communication happen between Microservices?

Answer:

Microservices communicate using **two main approaches**:

1. Synchronous Communication (REST APIs)

- Services interact using HTTP requests (e.g., `GET`, `POST`).
- **Problem:** If one service is down, the request fails.

Example:

```
@FeignClient(name = "order-service")
public interface OrderServiceClient {
    @GetMapping("/orders/{id}")
    Order getOrder(@PathVariable Long id);
}
```

> ***FeignClient*** *makes API calls easier.*

---

2. Asynchronous Communication (Message Queues)

- Uses **Kafka, RabbitMQ, or ActiveMQ** for event-driven communication.
- **Advantage:** Services are **loosely coupled** and do not depend on each other.

Example: Kafka Producer

```
@Autowired
private KafkaTemplate<String, String> kafkaTemplate;

public void sendMessage(String message) {
```

```
    kafkaTemplate.send("order-topic", message);
}
```

Asynchronous communication improves **scalability** and **resilience**.

6️ How do you handle authentication and authorization in Microservices?

Answer:

Spring Security and **JWT (JSON Web Tokens)** are commonly used for authentication.

Steps for JWT Authentication:

1️ **User logs in** → Server issues a **JWT token**
2️ **Client includes JWT in every request** (as `Authorization: Bearer <token>`) 3️ **Microservices validate JWT** using a shared secret key

Example: JWT Filter

```
public class JwtTokenFilter extends OncePerRequestFilter {
    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse
response, FilterChain chain)
            throws ServletException, IOException {
        String token = request.getHeader("Authorization");
        if (token != null && validateToken(token)) {
            chain.doFilter(request, response);
        } else {
            response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Invalid Token");
        }
    }
}
```

OAuth2 is another alternative for authentication in microservices.

7️ How do you implement logging and monitoring in Microservices?

Answer:

Microservices generate distributed logs, making debugging difficult. To solve this, tools like **ELK Stack (Elasticsearch, Logstash, Kibana)** and **Zipkin** (for tracing) are used.

How to enable centralized logging?

1️ Use **Spring Boot Actuator** for health and metrics:

```
management.endpoints.web.exposure.include=*
```

2️ Enable **Zipkin** for distributed tracing:

```
spring.zipkin.base-url=http://localhost:9411
```

This helps track requests across multiple services.

8️ How do you handle transactions across multiple microservices?

Answer:

Since each microservice has its own database, **transactions must be handled differently**.

1. Distributed Transactions (SAGA Pattern)

- A **saga** is a sequence of transactions, each triggering the next step.
- **Compensation Transactions** undo changes if a failure occurs.

Example of Choreography-based Saga:

- **Order Service** → Sends "Order Created" event
- **Payment Service** → Deducts payment and sends "Payment Successful"
- **Inventory Service** → Reduces stock

If any step fails, a **rollback event** compensates for previous actions.
