

## Spring Boot

### 1. Validations in Spring Boot:

In Spring Boot, validations are done using the Java Bean Validation framework, typically with Hibernate Validator as the default implementation. You define validation rules by adding annotations such as `@NotNull`, `@Size`, or `@Email` directly on the fields of your data transfer objects (DTOs). When a controller method receives input, you use the `@Valid` or `@Validated` annotation on the method parameter to tell Spring to automatically check these rules. If the input violates any validation constraint, Spring throws an exception like `MethodArgumentNotValidException`. To provide clear and consistent error responses, you can handle these exceptions globally using a class annotated with `@RestControllerAdvice` and methods annotated with `@ExceptionHandler`.

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

```
import jakarta.validation.constraints.*;

public class UserDTO {

    @NotNull(message = "Name is required")
    @Size(min = 2, max = 30, message = "Name must be 2 to 30 characters")
    private String name;

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

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

    // Getters and Setters
}
```

```
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.*;
import jakarta.validation.Valid;

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

    @PostMapping
    public String createUser(@Valid @RequestBody UserDTO user) {
        return "User created!";
    }
}
```

```

import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;

import java.util.HashMap;
import java.util.Map;

@RestControllerAdvice
public class GlobalExceptionHandler {

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

```

## 2. @RestControllerAdvice:

@RestControllerAdvice is a specialized annotation in Spring Boot used to handle exceptions globally across all @RestController classes.

It is a combination of:

- @ControllerAdvice (for global exception handling),
- and @ResponseBody (to return data as JSON/XML instead of a view).

## 3. @ControllerAdvice :

@ControllerAdvice is a special annotation in Spring that allows you to handle exceptions and apply common logic across multiple controllers in one place. It acts like a global interceptor for controllers.

When you annotate a class with @ControllerAdvice, you can define methods with @ExceptionHandler, @InitBinder, or @ModelAttribute annotations inside it. These methods will apply to all controllers (or a subset if you configure it) in your application.

## 4. Relaxed Binding:

Spring Boot uses some relaxed rules for binding Environment properties to @ConfigurationProperties beans, so there does not need to be an exact match between the Environment property name and the bean property name. Common examples where this is useful include dash-separated environment properties (for example, context-path binds to contextPath), and capitalized environment properties (for example, PORT binds to port).

```
@ConfigurationProperties(prefix = "my.main-project.person")
public class MyPersonProperties {

    private String firstName;

    public String getFirstName() {
        return this.firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

}
```

With the preceding code, the following properties names can all be used:

Table 3. relaxed binding

Property	Note
my.main-project.person.first-name	Kebab case, which is recommended for use in .properties and .yaml files.
my.main-project.person.firstName	Standard camel case syntax.
my.main-project.person.first_name	Underscore notation, which is an alternative format for use in .properties and .yaml files.
MY_MAINPROJECT_PERSON_FIRSTNAME	Upper case format, which is recommended when using system environment variables.

### Note

The `prefix` value for the annotation *must* be in kebab case (lowercase and separated by `-`, such as `my.main-project.person`).

Table 4. relaxed binding rules per property source

Property Source	Simple	List
Properties Files	Camel case, kebab case, or underscore notation	Standard list syntax using <code>[ ]</code> or comma-separated values
YAML Files	Camel case, kebab case, or underscore notation	Standard YAML list syntax or comma-separated values
Environment Variables	Upper case format with underscore as the delimiter (see <a href="#">Binding from Environment Variables</a> ).	Numeric values surrounded by underscores (see <a href="#">Binding from Environment Variables</a> )
System properties	Camel case, kebab case, or underscore notation	Standard list syntax using <code>[ ]</code> or comma-separated values

## 5. application.properties vs application.yml:

yml is more beneficial. Because it has Cleaner Structure, Less Repetition, Better Grouping, Useful for Complex Configs, More Human-Friendly.

If both .properties and .yml files exist Spring Boot merges them, but if the same key is present in both, .properties usually overrides .yml unless explicitly prioritized.

Means: whichever get loaded later it overrides the other one.

We can change the order of loading:

1. spring.config.location → most direct control over file priority:

```
--spring.config.location=classpath:/custom.yml,classpath:/application.properties
```

2. @PropertySource → manually load .properties into the context

```
@Configuration
@PropertySource("classpath:custom.properties")
public class CustomConfig {
}
```

This gives the loaded file lower priority than application.properties and application.yml

## 6. Order of loading:

Spring Boot uses a very particular PropertySource order that is designed to allow sensible overriding of values, properties are considered in the the following order:

1. Command line arguments.
2. Java System properties (System.getProperties()).
3. OS environment variables.
4. **@PropertySource** annotations on your **@Configuration** classes.
5. Application properties outside of your packaged jar (application.properties including YAML and profile variants).
6. Application properties packaged inside your jar (application.properties including YAML and profile variants).
7. Default properties (specified using SpringApplication.setDefaultProperties).

## 7. Spring Profiles:

In Spring Boot, profiles let you define different configurations for different environments — like dev, test, or prod — and load them automatically based on which profile is active.

## 8. @Profile:

@Profile is used to load specific beans only when a matching profile is active. It works with `spring.profiles.active`, which activates the environment.

## 9. What is cyclic dependency in Spring?

A cyclic dependency occurs when two or more beans depend on each other, leading to a circular reference that Spring cannot resolve during bean creation. This can lead to errors during application startup unless proxy-based dependency injection (like @Lazy) is used.

```
java

@Component
public class A {
    @Autowired
    private B b;
}

@Component
public class B {
    @Autowired
    private A a;
}

@Component
public class A {
    @Autowired
    @Lazy
    private B b;
}
```

## 10. Internal working of @Lazy:

Spring reads the @Lazy annotation on field B b.

It does not inject the real B into A immediately.

Instead, it injects a proxy object (like a shell) that acts like B but doesn't trigger its construction yet.

Spring creates A first (no problem, because it only injected a proxy for B).

Then it creates B and injects A into B as normal.

When any method of A is called Spring sees that the proxy for B is being used.

It now creates the real B bean (just-in-time).

Then it forwards the method call to the real object.

## 11. Advantages of singleton beans:

Spring creates the object only once, and whenever the application needs it, Spring reuses the same object instead of creating a new one.

This is useful because it saves memory (no need to create multiple copies), and the performance is better (object is ready to use). It's also easier to maintain because all parts of the application use the same instance, so if something changes in that object, it reflects everywhere.

Singleton is best when the bean does not hold any user-specific or request-specific data — like a logging service, database connector, or utility class.

## 12. Spring AOP:

AOP stands for Aspect-Oriented Programming.

It helps us separate cross-cutting concerns (like logging, security, transactions) from the main business logic.

Think of it as a way to inject behavior before, after, or around method calls, without changing the actual code.

**Aspect:** Class where you define extra behavior (e.g., logging, security).

**Advice:** The action taken by the aspect (e.g., code that runs before or after a method).

**Join Point:** A point in your program (usually a method) where advice can be applied.

**Pointcut:** Expression that matches join points (e.g., all methods in a package).

**Weaving:** Linking aspects to other parts of your code at runtime.

PaymentService



```
package com.example.demo.service;

import org.springframework.stereotype.Service;

@Service
public class PaymentService {

    public void makePayment() {
        System.out.println("Executing makePayment() - Business Logic");
    }

    public void refund() {
        System.out.println("Executing refund() - Business Logic");
    }
}
```

### Join Point:

These two methods (makePayment(), refund()) are join points—specific points where AOP code can run.

## X LoggingAspect



```
1 package com.example.demo.aspect;
2
3 import org.aspectj.lang.JoinPoint;
4 import org.aspectj.lang.annotation.Aspect;
5 import org.aspectj.lang.annotation.Before;
6 import org.springframework.stereotype.Component;
7
8 @Aspect // ✓ This class is an ASPECT (a module of cross-cutting concern)
9 @Component
10 public class LoggingAspect {
11
12     // ✓ ADVICE: This method runs before target methods → it's a @Before Advice
13     @Before("execution(* com.example.demo.service.PaymentService.*(..))")
14     public void logBefore(JoinPoint joinPoint) {
15
16         // ✓ JOIN POINT: Method currently being executed
17         String methodName = joinPoint.getSignature().getName();
18
19         // Logging the method name
20         System.out.println("🔍 Logging before method: " + methodName);
21     }
22 }
```

### Pointcut:

"execution(\* com.example.demo.service.PaymentService.\*(..))"

This is the pointcut expression. It matches all methods in PaymentService.

### Advice:

The logBefore() method is the advice. It runs before the matched method is executed.

### Weaving:

Spring AOP does weaving at runtime—it connects the LoggingAspect to PaymentService methods when the app runs.

```

1  package com.example.demo;
2
3  import com.example.demo.service.PaymentService;
4  import org.springframework.beans.factory.annotation.Autowired;
5  import org.springframework.boot.CommandLineRunner;
6  import org.springframework.boot.SpringApplication;
7  import org.springframework.boot.autoconfigure.SpringBootApplication;
8
9  @SpringBootApplication
10 public class DemoApplication implements CommandLineRunner {
11
12     @Autowired
13     private PaymentService paymentService;
14
15     public static void main(String[] args) {
16         SpringApplication.run(DemoApplication.class, args);
17     }
18
19     @Override
20     public void run(String... args) {
21         paymentService.makePayment(); // Triggers AOP
22         paymentService.refund();      // Triggers AOP
23     }
24 }

```

Application

### 13. Actuator:

Spring Boot Actuator is a built-in tool that provides ready-to-use endpoints to monitor and manage your Spring Boot application in production.

Feature	What It Does
/actuator/health	Shows app health (up/down)
/actuator/info	Shows custom info (version, name, etc.)
/actuator/metrics	Shows memory, CPU, GC, HTTP requests, etc.
/actuator/beans	Lists all Spring beans
/actuator/env	Shows environment properties (env, props)
/actuator/loggers	View or change logging levels at runtime
/actuator/mappings	Shows all URL endpoints in your app

### 14. How to handle multiple beans in spring boot?/@Qualifier and @Primary:

**@Primary** and **@Qualifier** can be used to resolve ambiguity when multiple beans of the same type are available for injection.

**@Primary** is used to mark one bean as the default when multiple beans of the same type exist. **@Qualifier** is used when you want to inject a specific bean by name.

\*\*\*



**@Primary** = "If you don't say anything, pick me."

**@Qualifier** = "I want this specific bean, no matter what."

## Car and Bike Components



```
@Component
public class Car implements Vehicle {
    public void start() {
        System.out.println("Car started");
    }
}

@Component
@Primary // ✅ This will be chosen by default
public class Bike implements Vehicle {
    public void start() {
        System.out.println("Bike started");
    }
}

@Component
public class VehicleService {

    @Autowired
    private Vehicle vehicle; // ✅ Injects Bike because it's marked @Primary

    public void startVehicle() {
        vehicle.start();
    }
}
```

Activate Windows  
Go to Settings to activate Windows.

```

@Component
✓ public class Car implements Vehicle {
✓     public void start() {
        System.out.println("Car started");
    }
}

@Component
✓ public class Bike implements Vehicle {
✓     public void start() {
        System.out.println("Bike started");
    }
}

@Component
✓ public class VehicleService {

    @Autowired
    @Qualifier("car") // ✓ Injects Car bean by name
    private Vehicle vehicle;

✓     public void startVehicle() {
        vehicle.start();
    }
}

```

**NB: @Qualifier can override @Primary**

**15. FixedDelay and FixedRate:**

**16. Cron Expression:**

**17. How can you validate two specific conditions in a YAML property file when creating a bean in Spring Boot?**

To validate multiple YAML properties when creating a Spring bean, you can use @ConditionalOnExpression.

```
import org.springframework.boot.autoconfigure.condition.ConditionalOnExpression;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class MyConfig {

    @Bean
    @ConditionalOnExpression(
        "'${myapp.featureA}' == 'true' and '${myapp.featureB}' == 'enabled'"
    )
    public MyService myService() {
        return new MyService();
    }
}
```

application.yml

yaml

```
myapp:
  featureA: true
  featureB: enabled
```

**NB:** Make sure to **wrap the \${} values in quotes ' ' to treat them as strings** in the expression.

## 18. Can you use multiple bean Configurations in Spring Boot?

Yes. Just have to use below annotation in main class

```
@ComponentScan(basePackages = {"com.yourapp", "com.other.configpackage"})
```

### **\*\*Other conditions:**

Here are simple examples for each conditional annotation shown in your image:

#### 1. @ConditionalOnProperty

**Use when:** You want to create a bean **only if a specific property is set**.

**application.yml:**

```
app.feature.enabled: true
```

## Java:

```
@Bean
@ConditionalOnProperty(name = "app.feature.enabled", havingValue = "true")
public FeatureService featureService() {
    return new FeatureService();
}
```

---

## ✅ 2. @ConditionalOnClass

**Use when:** You want to create a bean **only if a specific class is present** in the classpath.

## Java:

```
@Bean
@ConditionalOnClass(name = "com.mysql.cj.jdbc.Driver")
public DataSource mysqlDataSource() {
    return new HikariDataSource();
}
```

---

## ✅ 3. @ConditionalOnMissingBean

**Use when:** You want to create a bean **only if a bean of the same type does NOT** already exist.

## Java:

```
@Bean
@ConditionalOnMissingBean(UserService.class)
public UserService userService() {
    return new DefaultUserService();
}
```

---

## ✅ 4. @ConditionalOnBean

**Use when:** You want to create a bean **only if another bean already exists**.

## Java:

```
@Bean
@ConditionalOnBean(UserRepository.class)
public UserReportService userReportService() {
    return new UserReportService();
}
```

---

## ✅ 5. @ConditionalOnExpression

**Use when:** You want to create a bean **based on SpEL (Spring Expression Language)** conditions.

**application.yml:**

```
app.mode: prod
```

**Java:**

```
@Bean
@ConditionalOnExpression("'${app.mode}' == 'prod'")
public ProdOnlyService prodOnlyService() {
    return new ProdOnlyService();
}
```

---

## ✅ 6. @ConditionalOnMissingClass

**Use when:** You want to create a bean **only if a class is NOT on the classpath**.

**Java:**

```
@Bean
@ConditionalOnMissingClass("com.external.LibraryX")
public BackupService backupService() {
    return new BackupService();
}
```

---

## 📌 Summary Table

Annotation	Condition	Example Use Case
@ConditionalOnProperty	Property value check	Feature toggle
@ConditionalOnClass	Class exists	Load bean if JDBC driver is present
@ConditionalOnMissingBean	No existing bean of type	Provide fallback/default bean
@ConditionalOnBean	Bean exists	Dependent bean setup
@ConditionalOnExpression	SpEL expression is true	Complex logic using properties
@ConditionalOnMissingClass	Class is absent	Avoid loading beans if library is missing

---

Let me know if you'd like me to bundle these into a single runnable Spring Boot project!

**\*\*Custom condition:**


**create a bean only if the current day is Monday.**

```
import org.springframework.context.annotation.Condition;
import org.springframework.context.annotation.ConditionContext;
import org.springframework.core.type.AnnotatedTypeMetadata;

import java.time.DayOfWeek;
import java.time.LocalDate;

public class MondayCondition implements Condition {

    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
        // Only return true if today is Monday
        return LocalDate.now().getDayOfWeek() == DayOfWeek.MONDAY;
    }
}
```



```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Conditional;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfig {

    @Bean
    @Conditional(MondayCondition.class)
    public String mondayBean() {
        return "Hello, it's Monday!";
    }
}
```



```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class GreetingService {

    @Autowired(required = false)
    private String mondayBean;

    public void greet() {
        if (mondayBean != null) {
            System.out.println(mondayBean);
        } else {
            System.out.println("Not Monday today.");
        }
    }
}

```

#### 19. How to make a bean optional?

@Autowired(required = false)

#### 20. Suppose your spring application is experiencing performance issues under high load. What steps would you take to diagnose and resolve the issue?

We can use different types of profiling and monitoring for real time metrics(memory, CPU etc.)

##### Types of Profiling to Use:

- **CPU Profiling** – Find methods using too much CPU.
- **Thread & Memory Profiling** – Detect memory leaks, thread locks.
- **Database Profiling** – Spot slow/unindexed queries.
- **Log Analysis** – Identify hidden issues or patterns.
- **Real-Time Monitoring** – Get alerts as issues occur.
- **Transaction Tracing** – Track a request across services to locate delays.

##### Tools to Use:

- **JProfiler** – Analyze CPU and memory usage.
- **VisualVM** – Live monitoring of memory and threads.
- **Spring Boot Actuator + Micrometer** – Get application metrics.
- **Zipkin** – Trace requests across microservices.

## 21. How to define custom annotation in spring boot?

You can define a custom annotation in Spring Boot when you want to apply reusable logic across multiple classes or methods, such as logging, validation, or security. You define it using `@interface`, and then use AOP or custom logic to handle its behavior at runtime.

eg.

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface LogExecutionTime {
}
```

## 22. what is Caching in spring boot?

Caching in Spring Boot is used to **store the result of method calls** so that repeated calls with the same input return the stored result instead of recalculating or hitting the database again. This helps in **improving performance** and **reducing load** on services or databases.

Spring Boot provides built-in support for caching using annotations like `@Cacheable`, `@CachePut`, and `@CacheEvict`. You can use in-memory caches like `ConcurrentHashMap`, or integrate with external ones like Redis, Ehcache, etc.

```
@SpringBootApplication
@EnableCaching
public class MyApp { }
```

### 2. Add caching to a method:

```
java

@Service
public class UserService {

    @Cacheable("users")
    public User getUserById(Long id) {
        System.out.println("Fetching from DB...");
        return userRepository.findById(id).orElse(null);
    }
}
```

## 23. How inter communication happens in microservices? give example with RestTemplate with eureka

Microservices communicate either synchronously (via HTTP using REST clients) or asynchronously (using messaging systems like Kafka, RabbitMQ).



Using **Eureka** for service discovery, microservices don't need to know exact URLs — they call each other using **service names**, and Eureka resolves the correct instance.

```
@Configuration
public class AppConfig {
    @Bean
    @LoadBalanced // Enables Eureka-based load balancing
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

```
@Service
public class OrderService {
    @Autowired
    private RestTemplate restTemplate;

    public User getUserById(Long id) {
        return restTemplate.getForObject("http://USER-SERVICE/users/" + id, User.class);
    }
}
```

## 24. How can we customize automatic configuration in spring boot?

Spring Boot comes with auto-configuration, which configures components based on the dependencies in the classpath.

You can customize it in several ways:

- Override default beans using `@Bean`
- Use `@ConditionalOnMissingBean` to let your beans take priority
- Use `application.properties` or `application.yml` to override settings
- Exclude auto-config classes using `@SpringBootApplication(exclude = ...)`

```
@Bean
@ConditionalOnMissingBean
public DataSource myDataSource() {
    return new HikariDataSource(); // Custom data source
}
```

```

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

```

## 25. Explain different scopes available in spring boot?

Spring Boot supports multiple bean scopes:

Scope	Description
singleton	(Default) One instance per container
prototype	New instance each time it's requested
request	One instance per HTTP request (Web)
session	One per HTTP session (Web)
application	One per ServletContext
websocket	One per WebSocket session

## 26. How to create custom scope in spring boot?

To create a custom scope, implement the `org.springframework.beans.factory.config.Scope` interface and register it using `CustomScopeConfigurer`.

### Custom Scope Implementation:

```
java

public class MyCustomScope implements Scope {
    private Map<String, Object> beans = new HashMap<>();

    @Override
    public Object get(String name, ObjectFactory<?> objectFactory) {
        return beans.computeIfAbsent(name, k -> objectFactory.getObject());
    }

    @Override
    public Object remove(String name) {
        return beans.remove(name);
    }

    // Other methods can be left empty or customized
}
```

### Registering the Scope:

```
java

@Configuration
public class ScopeConfig {
    @Bean
    public static CustomScopeConfigurer configurer() {
        CustomScopeConfigurer configurer = new CustomScopeConfigurer();
        configurer.addScope("custom", new MyCustomScope());
        return configurer;
    }
}
```

```
@Component
@Scope("custom")
public class MyBean {
    // This bean will follow custom scope
}
```

### 27. What is the difference between service registry and service discovery?

In a **microservices architecture**, services need to find and communicate with each other. Since services can scale up/down or change IPs, we can't hardcode their locations. This is where **Service Registry** and **Service Discovery** come into play.

## 1. Service Registry

- It's like a **phone book** for microservices.
- It **stores the network locations** (host and port) of each service instance.
- When a service starts, it **registers itself** with the registry.

**Example tools:** Eureka Server, Consul, Zookeeper

## 2. Service Discovery

- It's the process of **looking up** a service's location using its name.
- A service (or a load balancer) queries the registry to **discover** the address of another service.

Can be of two types:

- **Client-side discovery:** Client gets instance list and makes request (e.g., Netflix Eureka with RestTemplate)
- **Server-side discovery:** A proxy (e.g., Zuul or API Gateway) handles discovery

## 28. explain saga design pattern

The **Saga Pattern** is a **microservices transaction management pattern** used to **maintain data consistency** across multiple services **without using distributed transactions**.

Instead of locking resources across services, Saga breaks a transaction into a **series of local transactions**. Each service performs its own part and either proceeds or **compensates** (rolls back) based on success or failure.

## 29. Explain Idempotency in microservices.

**Idempotency** means that **making the same request multiple times** will have the **same effect** as making it once. It's a key concept in microservices to ensure **safety, consistency, and fault tolerance** — especially in scenarios involving retries or network issues.

Without idempotency, **the same operation could be processed multiple times**, leading to incorrect results — like **multiple payments, duplicate records**, etc.

**GET, PUT, DELETE** → are **idempotent**

**POST** → is **not idempotent** (but can be made idempotent)

## 30. how to achieve loose coupling between two microservices services?

**Loose coupling** means microservices are **independent** — changes in one service don't break or tightly affect others. This is **essential** for scalability, independent deployment, and better fault tolerance in microservices architecture.

Techniques to Achieve Loose Coupling

### 1. Use REST APIs with well-defined contracts

### 2. Use Asynchronous Communication (Event-driven) like kafka

### 3. Use API Gateway

### 4. Service Discovery with Eureka

## 5. Avoid Direct Database Sharing

## 6. Use Interface-based Communication

### 31. Explain fault tolerance

**Fault tolerance** is the ability of a system to continue operating properly **even when one or more of its components fail**.

In a **microservices architecture**, where many services interact over the network, failures are inevitable — services may go down, become slow, or return errors. A fault-tolerant system **detects these failures** and handles them **gracefully** to maintain overall system reliability and availability.

Without fault tolerance:

- A single failed service can **crash** the entire application.
- **Cascading failures** may happen when one service's failure causes others to fail.

### Common Fault Tolerance Techniques

#### 1. Circuit Breaker

Stops calling a failing service for a period of time. Prevents repeated failures from overwhelming the system.



Tools: **Resilience4j**, **Hystrix** (legacy)

```
java
CopyEdit
@CircuitBreaker(name = "inventoryService", fallbackMethod = "fallbackInventory")
public Product getProduct() {
    return restTemplate.getForObject("http://INVENTORY-SERVICE/product",
    Product.class);
}

public Product fallbackInventory(Throwable t) {
    return new Product("Default", 0);
}
```

---

#### 2. Retries

Automatically retry the request a few times before failing.

```
java
CopyEdit
@Retry(name = "paymentService", maxAttempts = 3)
public PaymentResponse pay() {
    // call external service
}
```

---

#### 3. Timeouts

Avoid waiting indefinitely for a response. Define a maximum wait time.

```
properties
CopyEdit
```

```
# application.yml
resilience4j.retry.instances.paymentService.waitDuration=2s
```

---

#### 4. Fallback

Provide an alternative response or default behavior if the main service fails.

```
java
CopyEdit
public Product fallbackInventory(Throwable t) {
    // return a default product if service fails
    return new Product("Fallback Product", 0);
}
```

---

#### 5. Bulkhead Pattern

Isolate failures by partitioning system resources. One service's failure won't affect others.

Example: Separate thread pools for different services.

##### 32. Difference between hashmap and concurrent hashmap.

HashMap is **faster** in single-threaded environments. It allows one null key and multiple null values

ConcurrentHashMap is **optimized for multi-threading**, using **bucket-level locking** (a technique called **lock striping**) instead of locking the entire map like `Collections.synchronizedMap()`. It doesn't support null values or keys

##### 33. How will you remove duplicates and maintain the insertion order from duplicate array?

To remove duplicates **while maintaining insertion order**, you can use **LinkedHashSet** in Java.

##### 34. Explain Volatile keyword.

The `volatile` keyword is used in Java to make **a variable always visible to all threads**.

Normally, when multiple threads use a shared variable, each thread may keep a **copy of the variable in its own memory** (cache). So if one thread changes the variable, the others may still see the old value.

When a variable is marked as `volatile`, Java ensures that:

- **Every read** of the variable gets the **latest value from main memory**
- **Every write** to the variable is **immediately visible** to other threads

##### 35. Explain vertical scaling and horizontal scaling

When your application needs to handle **more load** (like more users or data), you can scale it in two main ways:

Vertical: Moving from a 4-core server to a 16-core server, Increasing RAM from 8 GB to 32 GB

Horizontal:

You add **more machines** (or instances) to handle the load.

You're running **the same application** on multiple servers.

**Example:**

- Deploying your Spring Boot app on 3 servers behind a load balancer
- Adding more pods in Kubernetes

### 36. == vs .equals

== Checks if **both references point to the exact same object** in memory. It does **not** compare the content.

.equals Compares the **actual content or values** inside the objects.