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.

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

```java
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
}
```

```java
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!";
    }
}
```

```java
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 `.yml` 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 `.yml` 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 `[ ]` 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 Binding from Environment Variables).	Numeric values surrounded by underscores (see Binding from Environment Variables)
System properties	Camel case, kebab case, or underscore notation	Standard list syntax using `[ ]` 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.