Tab 1

# **Spring Boot Validations**

Spring Boot provides built-in support for validation using **Jakarta Bean Validation (formerly Javax)** and integrates with **Spring’s @Valid and @Validated annotations**.

### **1. Key Annotations for Validation**

**Basic Constraints (Jakarta Bean Validation)**

* @NotNull – Field must not be null.
* @NotEmpty – Field must not be null and must contain at least one character.
* @NotBlank – Field must not be null and must contain at least one **non-whitespace** character.
* @Size(min, max) – Restricts string length or collection size.
* @Min(value) – Minimum numeric value allowed.
* @Max(value) – Maximum numeric value allowed.
* @Positive – Must be greater than zero.
* @PositiveOrZero – Must be zero or greater.
* @Negative – Must be less than zero.
* @NegativeOrZero – Must be zero or less.
* @Email – Ensures a valid email format.
* @Pattern(regexp) – Validates against a regex pattern.
* @Past – Date must be in the past.
* @PastOrPresent – Date must be in the past or present.
* @Future – Date must be in the future.
* @FutureOrPresent – Date must be in the future or present.

**Advanced Constraints**

* @Digits(integer, fraction) – Ensures the numeric field has a specific integer and fraction length.
* @DecimalMin(value, inclusive) – Minimum decimal value.
* @DecimalMax(value, inclusive) – Maximum decimal value.
* @AssertTrue – Field must be true.
* @AssertFalse – Field must be false.

### **2. Applying Validations in DTOs**

import jakarta.validation.constraints.\*;

public class CustomerDTO {

@NotNull(message = "Customer name cannot be null")

@Size(min = 3, max = 50, message = "Customer name must be between 3 and 50 characters")

private String name;

@NotBlank(message = "Email cannot be blank")

@Email(message = "Invalid email format")

private String email;

@Pattern(regexp = "\\d{10}", message = "Phone number must be 10 digits")

private String phoneNumber;

@Min(value = 18, message = "Age must be at least 18")

private int age;

@Past(message = "Date of birth must be in the past")

private LocalDate dob;

}

### **3. Validating DTOs in Controllers**

Use @Valid for **single object validation**.

@RestController

@RequestMapping("/customers")

public class CustomerController {

@PostMapping

public ResponseEntity<String> createCustomer(@Valid @RequestBody CustomerDTO customerDTO) {

return ResponseEntity.ok("Customer created successfully");

}

}

### **4. Handling Validation Errors Using @ExceptionHandler**

You can capture validation errors globally using @ControllerAdvice.

**@RestControllerAdvice**

public class GlobalExceptionHandler {

**@ExceptionHandler(MethodArgumentNotValidException.class)**

public ResponseEntity<Map<String, String>> handleValidationExceptions(**MethodArgumentNotValidException ex**) {

**Map<String, String> errors = new HashMap<>();**

**ex.getBindingResult().getFieldErrors().forEach(error ->**

**errors.put(error.getField(), error.getDefaultMessage())**

**);**

return ResponseEntity.badRequest().body(**errors**);

}

}

### **5. Custom Validations**

Creating a Custom Validator

1. Create an annotation:

@Constraint(validatedBy = PanCardValidator.class)

@Target({ElementType.FIELD})

@Retention(RetentionPolicy.RUNTIME)

public @interface ValidPan {

String message() default "Invalid PAN number";

Class<?>[] groups() default {};

Class<? extends Payload>[] payload() default {};

}

1. Implement the logic:

public class PanCardValidator implements ConstraintValidator<**ValidPan**, String> {

**private static final String PAN\_REGEX = "[A-Z]{5}[0-9]{4}[A-Z]{1}";**

@Override

public boolean isValid(String value, ConstraintValidatorContext context) {

return value != null && value.matches(**PAN\_REGEX**);

}}

1. Use the annotation in DTO:

public class UserDTO {

@ValidPan

private String panNumber;

}

### **6. Validating Collections or Nested Objects**

Use @Valid inside the DTO to validate nested objects or collections.

public class BankDTO {

@NotBlank

private String bankName;

@Valid

@NotEmpty

private List<**@Valid** BranchDTO> branches;

}

### **7. Using @Validated for Method-Level Validation**

Apply @Validated on the class level to enable method-level validations.

@Service

@Validated

public class BankService {

public void updateBankName(@NotBlank String name) {

// Business logic

}

}

### **8. Enabling Validation in Spring Boot**

Spring Boot **automatically** enables validation when spring-boot-starter-validation is included:

**<dependency>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-starter-validation</artifactId>**

**</dependency>**

### **Conclusion**

* Use @Valid in controllers to validate request payloads.
* Use @Validated in services for method-level validation.
* Handle errors using @ControllerAdvice for a structured error response.
* Create custom validators when built-in constraints are insufficient.
* Validate nested objects and collections using @Valid.