### Validating User Input in Spring Boot (using `@Valid`, `@NotNull`, etc.)

Spring Boot provides robust validation support for user input using JSR-303/JSR-380 Bean Validation API annotations (part of Java EE). This allows you to validate input from forms, APIs, or any object that needs validation.

### Key Concepts:

- \*\*JSR-303/JSR-380\*\*: Bean validation specification that defines a standard way to validate Java beans using annotations like `@NotNull`, `@Size`, `@Min`, etc.

- \*\*`@Valid`\*\*: Triggers validation on method parameters or fields.

- \*\*`@NotNull`, `@NotEmpty`, `@Size`, etc.\*\*: Validation annotations used to enforce specific rules on fields or method parameters.

- \*\*`BindingResult`\*\*: Used to capture validation errors and provide feedback.

---

### Common Validation Annotations

- \*\*`@NotNull`\*\*: Ensures that a field is not null.

- \*\*`@NotEmpty`\*\*: Ensures that a field is not null and not empty (applicable to collections and strings).

- \*\*`@NotBlank`\*\*: Ensures that a string is not null, not empty, and contains non-whitespace characters.

- \*\*`@Size`\*\*: Specifies the size constraints for strings, arrays, or collections.

- \*\*`@Min`\*\*: Ensures that a number is at least a certain value.

- \*\*`@Max`\*\*: Ensures that a number does not exceed a certain value.

- \*\*`@Pattern`\*\*: Ensures that a string matches a given regular expression pattern.

- \*\*`@Email`\*\*: Validates that a string is in a valid email format.

- \*\*`@Past`, `@Future`, `@PastOrPresent`, `@FutureOrPresent`\*\*: Validate date or time values.

---

### Using Validation in Spring Boot

#### 1. \*\*Setting Up Validation Dependency\*\*

If you are using Maven, Spring Boot already includes `spring-boot-starter-validation`, which provides everything you need for validation.

Make sure your `pom.xml` includes this:

```xml

<dependency>

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

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

</dependency>

```

---

#### 2. \*\*Creating a DTO (Data Transfer Object) with Validation Annotations\*\*

Here’s an example of a `User` DTO where you can use validation annotations to ensure proper input.

```java

import javax.validation.constraints.Email;

import javax.validation.constraints.NotBlank;

import javax.validation.constraints.NotNull;

import javax.validation.constraints.Size;

public class UserDTO {

@NotNull(message = "Id cannot be null")

private Long id;

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

@Size(min = 2, max = 30, message = "Name must be between 2 and 30 characters")

private String name;

@Email(message = "Email should be valid")

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

private String email;

// Getters and Setters

}

```

In this example:

- The `id` field is required to be non-null.

- The `name` field must not be blank, and it should be between 2 and 30 characters.

- The `email` field must be a valid email format and cannot be blank.

---

#### 3. \*\*Validating Input in a REST Controller\*\*

You can validate incoming data in a Spring Boot controller by using the `@Valid` or `@Validated` annotations on the method parameters.

```java

import org.springframework.http.ResponseEntity;

import org.springframework.validation.BindingResult;

import org.springframework.web.bind.annotation.PostMapping;

import org.springframework.web.bind.annotation.RequestBody;

import org.springframework.web.bind.annotation.RestController;

import javax.validation.Valid;

@RestController

public class UserController {

@PostMapping("/users")

public ResponseEntity<String> createUser(@Valid @RequestBody UserDTO userDTO, BindingResult result) {

if (result.hasErrors()) {

return ResponseEntity.badRequest().body(result.getAllErrors().toString());

}

return ResponseEntity.ok("User is valid and created successfully");

}

}

```

#### Key Points:

- The `@Valid` annotation triggers validation on the `UserDTO` object.

- The `BindingResult` parameter captures validation errors. If there are errors, you can handle them and return appropriate feedback (e.g., HTTP 400 Bad Request).

---

### Handling Validation Errors Globally

You can centralize the handling of validation errors using `@ControllerAdvice` to avoid writing the same error-handling code in each controller.

#### Example of Global Exception Handling for Validation Errors:

```java

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.validation.FieldError;

import org.springframework.web.bind.MethodArgumentNotValidException;

import org.springframework.web.bind.annotation.ControllerAdvice;

import org.springframework.web.bind.annotation.ExceptionHandler;

import org.springframework.web.bind.annotation.ResponseStatus;

import java.util.HashMap;

import java.util.Map;

@ControllerAdvice

public class GlobalExceptionHandler {

@ResponseStatus(HttpStatus.BAD\_REQUEST)

@ExceptionHandler(MethodArgumentNotValidException.class)

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

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

ex.getBindingResult().getAllErrors().forEach((error) -> {

String fieldName = ((FieldError) error).getField();

String errorMessage = error.getDefaultMessage();

errors.put(fieldName, errorMessage);

});

return new ResponseEntity<>(errors, HttpStatus.BAD\_REQUEST);

}

}

```

#### Explanation:

- \*\*`@ControllerAdvice`\*\*: This annotation is used to handle exceptions globally.

- \*\*`MethodArgumentNotValidException`\*\*: This exception is thrown when a `@Valid` object fails validation.

- This handler extracts validation errors and returns them as a `400 Bad Request` with a detailed error message for each field.

---

### Custom Validation Annotations

In some cases, the standard annotations won’t be enough, and you’ll need to create custom validations. You can define custom annotations by implementing `ConstraintValidator`.

#### Example of a Custom Validator for a Password Field:

1. \*\*Create the Custom Annotation\*\*:

```java

import javax.validation.Constraint;

import javax.validation.Payload;

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

@Constraint(validatedBy = PasswordValidator.class)

@Target({ ElementType.METHOD, ElementType.FIELD })

@Retention(RetentionPolicy.RUNTIME)

public @interface ValidPassword {

String message() default "Invalid password";

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

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

}

```

2. \*\*Implement the Custom Validator\*\*:

```java

import javax.validation.ConstraintValidator;

import javax.validation.ConstraintValidatorContext;

public class PasswordValidator implements ConstraintValidator<ValidPassword, String> {

@Override

public void initialize(ValidPassword constraintAnnotation) {

}

@Override

public boolean isValid(String password, ConstraintValidatorContext context) {

return password != null && password.length() >= 8 && password.matches(".\*\\d.\*");

}

}

```

In this example, the custom validator checks that the password is at least 8 characters long and contains at least one digit.

3. \*\*Using the Custom Annotation in a DTO\*\*:

```java

public class UserDTO {

@ValidPassword

private String password;

// Other fields, getters, setters

}

```

---

### Example: Full REST API with Input Validation

Here’s a full example demonstrating how to validate user input in a Spring Boot REST API:

1. \*\*User DTO\*\*:

```java

import javax.validation.constraints.Email;

import javax.validation.constraints.NotBlank;

import javax.validation.constraints.Size;

public class UserDTO {

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

@Size(min = 2, max = 30, message = "Name must be between 2 and 30 characters")

private String name;

@Email(message = "Email should be valid")

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

private String email;

@ValidPassword

private String password;

// Getters and Setters

}

```

2. \*\*Controller\*\*:

```java

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.PostMapping;

import org.springframework.web.bind.annotation.RequestBody;

import org.springframework.web.bind.annotation.RestController;

import javax.validation.Valid;

@RestController

public class UserController {

@PostMapping("/users")

public ResponseEntity<String> createUser(@Valid @RequestBody UserDTO userDTO) {

return ResponseEntity.ok("User is valid and created successfully");

}

}

```

3. \*\*Global Exception Handler\*\* (Optional, for better error messages):

```java

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.validation.FieldError;

import org.springframework.web.bind.MethodArgumentNotValidException;

import org.springframework.web.bind.annotation.ControllerAdvice;

import org.springframework.web.bind.annotation.ExceptionHandler;

import org.springframework.web.bind.annotation.ResponseStatus;

import java.util.HashMap;

import java.util.Map;

@ControllerAdvice

public class GlobalExceptionHandler {

@ResponseStatus(HttpStatus.BAD\_REQUEST)

@ExceptionHandler(MethodArgumentNotValidException.class)

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

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

ex.getBindingResult().getAllErrors().forEach((error) -> {

String fieldName = ((FieldError) error).getField();

String errorMessage = error.getDefaultMessage();

errors.put(fieldName, errorMessage

);

});

return new ResponseEntity<>(errors, HttpStatus.BAD\_REQUEST);

}

}

```

---

### Conclusion

- Spring Boot simplifies input validation using JSR-303/380 Bean Validation annotations.

- Common annotations like `@NotNull`, `@NotBlank`, and `@Size` can be used to enforce input rules.

- The `@Valid` annotation triggers validation, and `BindingResult` can be used to capture and handle validation errors.

- Centralizing error handling with `@ControllerAdvice` improves code maintainability.

- You can create custom validation annotations for specialized use cases.

This setup ensures that input to your application is validated, preventing invalid data from being processed.