**Custom Annotations and Validations**

In Spring Boot, custom annotations can be created to add metadata to classes, methods, or fields in your application. These annotations can then be processed at runtime using reflection or other mechanisms to perform specific actions or configurations.

**Demo Project was done to illustrate custom annotations.**

**Dependencies used for the project:**

<dependency>

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

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>8.0.33</version>

</dependency>

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<optional>true</optional>

</dependency>

<dependency>

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

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

<scope>test</scope>

</dependency>

**Validation using Default Annotations**

Firstly, we should check the validation of data using default annotations in Spring boot.

User Request Dao Class:

The below default annotations were used for data validation.

* @NotNull- The field should not be null when we are passing through POST operation
* @Pattern (regexp = "^\\d {10} $", message = "invalid mobile number entered ") –

^: Asserts the start of the string.

\d: Matches any digit (equivalent to [0-9]).

{10}: Specifies that exactly 10 occurrences of the preceding \d (digit) pattern is expected.

$: Asserts the end of the string.

The user input of Mobile Number should match with the above condition of regex expression or else the error message specified will be displayed.

* @NotBlank – The input field should not be blank when we are passing through post operation
* @Min (18)

@Max (60)

private int age; - The age should be minimum of 18 years and maximum of 60 years

* @Email- There should be some fields before @gmail.com or @yahoo.com when giving the email as input

public class UserRequest {

@NotNull (message = "username shouldn't be null")

private String name;

@Pattern (regexp = "^\\d{10}$",message = "invalid mobile number entered ")

private String mobile;

@NotNull (message = "gender shouldn't be null")

private String gender;

@Min (18)

@Max (60)

private int age;

@NotBlank

private String nationality;

@Email (message = "invalid email address")

@ValidateUniqueEmail

private String email;

//custom annotation

@ValidateUserType

private String userType; //permanent or vendor

@ValidatePinCodeLength

private String pincode; //

}

UserService Class:

This method will save the user with the help of userRepository

@Service

public class UserService {

@Autowired

private UserRepository repository;

public User saveUser(UserRequest userRequest) {

User user = User.

build (0, userRequest.getName(), userRequest.getEmail(),

userRequest.getMobile(), userRequest.getGender(), userRequest.getAge(), userRequest.getNationality(), userRequest.getUserType(),userRequest.getPincode());

return repository.save(user);

}

UserController Class:

The saveUser method is mapped to the "/signup" endpoint . @Valid is used to validate the userRequest and the exception is handled through the ApplicationExceptionHandler class.

@PostMapping("/signup")

public ResponseEntity<User> saveUser(@RequestBody @Valid UserRequest userRequest){

return new ResponseEntity<>(service.saveUser(userRequest), HttpStatus.CREATED);

}

ApplicationExceptionHanlder Class:

package com.demo.api.advice;

import org.springframework.http.HttpStatus;

import org.springframework.web.bind.MethodArgumentNotValidException;

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

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

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

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

import com.demo.api.exception.UserNotFoundException;

import java.util.HashMap;

import java.util.Map;

@RestControllerAdvice

public class ApplicationExceptionHandler {

@ResponseStatus (HttpStatus.BAD\_REQUEST)

//If the method which is annotated with @Valid throws exception,this method should execute

@ExceptionHandler (MethodArgumentNotValidException.class)

public Map<String, String> handleInvalidArgument(MethodArgumentNotValidException ex) {

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

//Get the binding errors and its list of errors>Iterate through them and get the error field and its message

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

errorMap.put(error.getField(), error.getDefaultMessage());

});

return errorMap;

}

//It will return the map that contains "errorMessage" and its actual error message

@ResponseStatus (HttpStatus.INTERNAL\_SERVER\_ERROR)

@ExceptionHandler (UserNotFoundException.class)

public Map<String, String> handleBusinessException(UserNotFoundException ex) {

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

errorMap.put("errorMessage", ex.getMessage());

return errorMap;

}

}

**Validation using Custom Annotations**

The custom annotations can be defined using @interface in Springboot.

* Validating the userType

Step 1: Define the annotation class. Here we are checking whether the given userType is valid or not i.e. whether he is permnent or contractual.

Validate userType Class:

@Target ({ElementType.FIELD,ElementType.PARAMETER}) //where u want to use it exactly

//It specifies that the annotated annotation should be retained by the Java Virtual Machine (JVM) so that it can be accessed at runtime via reflection

@Retention (RetentionPolicy.RUNTIME) //At what time u want to execute

@Constraint (validatedBy=UserTypeValidator.class) //where the validation logic is written

public @interface ValidateUserType {

public String message ()

default "Invalid user Type: It should either Permanent or Vendor";

//These methods allow users of the @NotNull annotation to specify custom validation groups

//and payloads when using this constraint.

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

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

}

Step 2: Implement the logic through which u are checking the valid user or not in the other class.

Here the userTypes list contains only two types of userTypes and they are “Permanent” and “Vendor”. The given input should either “Permanent” or “Vendor” otherwise, it will throw the error message

public class UserTypeValidator implements ConstraintValidator<ValidateUserType, String> {

@Override

public boolean isValid(String userType, ConstraintValidatorContext context) {

List<String> userTypes = Arrays.asList("Permanent","Vendor");

return userTypes.contains(userType);

} }

Step3: The parameter of Dao class should be annotated with the Custom Annotation name

@ValidateUserType

private String userType;

* Validating the pincode length

Step 1: Define the annotation class. Here we are checking the pincode length of the user.

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

@Retention (RetentionPolicy.RUNTIME)

@Constraint (validatedBy=UserPinCodeLengthValidator.class)

public @interface ValidatePinCodeLength {

public String message () default "Invalid pincode:The length of the pincode should be 5";

//These methods allow users of the @NotNull annotation to specify custom validation groups

//and payloads when using this constraint.

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

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

}

Step2: Implement the logic through which u are checking the length of the pincode.

public class UserPinCodeLengthValidator implements ConstraintValidator<ValidatePinCodeLength, String>

{

UserRequest userRequest;

@Override

public boolean isValid(String pincode, ConstraintValidatorContext context) {

if (pincode.length()==5)

{

return true;

}

return false;

}

}

Step3: The parameter of Dao class should be annotated with the Custom Annotation name.

@ValidatePinCodeLength

private String pincode;

* Validating the unique email address

Step1: Define the annotation class. Here we are checking whether the given user email address is unique or not.

@Target ({ElementType.FIELD,ElementType.PARAMETER,ElementType.TYPE})

@Retention (RetentionPolicy.RUNTIME)

@Constraint (validatedBy=UniqueEmailValidator.class)

public @interface ValidateUniqueEmail {

public String message () default "Email should be unique";

//These methods allow users of the @NotNull annotation to specify custom validation groups

//and payloads when using this constraint.

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

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

}

Step2: Implement the logic through which u are checking whether the given email id is unique or not.

public class UniqueEmailValidator implements ConstraintValidator<ValidateUniqueEmail, String> {

@Autowired

UserRepository userRepository;

@Override

public boolean isValid(String value, ConstraintValidatorContext context) {

//If the given email id is already in the list, then return zero

if(userRepository.findByEmail(value).size()==0)

{

return true;

}

return false;

}

}

Step3: The parameter of Dao class should be annotated with the Custom Annotation name.

@ValidateUniqueEmail

private String email;

If the wrong input is given, then this error message is displayed as per below.

