**Validation of Spring Boot RESTful Web Services with JSR 380**

Hibernate Validator with JSR 380 – Bean Validation 2.0

Our Spring Boot RESTful Web Service requires an artifact spring-boot-starter-web. It will transitively add a dependency for Hibernate Validator 6.0.7, which is a reference implementation for Bean Validation 2.0 or [JSR 380](https://jcp.org/en/jsr/detail?id=380).

Bean Validation 2.0 includes some additional features like:

* Apply validation constraints to collection’s type declaration, like List<@NotNull String> values
* Support for Java 8 Optional
* Support for the new date/time data types for @Past  and @Future
* New built-in constraints like: @Email, @Positive, @PositiveOrZero, @Negative,  etc
* All built-in constraints are marked as repeatable

We are going to use some of them in our example. Let’s jump to the code now.

Bean with Validation annotations

Our example is a simple Spring RESTFul Web Service for saving an Employee to the database. It uses a DTO (Data Transfer Object) as a request bean. This request bean is then transformed into Entity object to perform database operations. Here we are separating the layers by not exposing Entity classes to the external world.

Our Employee object needs to satisfy following conditions:

1. An employee should have a first name, last name, and designation (all mandatory fields)
2. Employee’s salary should be a positive number
3. Employee’s date of birth should be in past
4. An employee can have multiple email addresses with a valid format

To fulfill this requirement our request bean EmployeeDO  should use following validation annotations:

EmployeeDO.java

Java



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | package com.example.restful.dto;    import java.io.Serializable;  import java.util.Date;  import java.util.List;    import javax.validation.constraints.Email;  import javax.validation.constraints.NotEmpty;  import javax.validation.constraints.NotNull;  import javax.validation.constraints.Past;  import javax.validation.constraints.Positive;    import com.fasterxml.jackson.annotation.JsonFormat;    public class EmployeeDO implements Serializable {    private static final long serialVersionUID = -5662688722276730214L;    private Long employeeId;    @NotEmpty(message = "{employee.firstName.required}")  private String firstName;    @NotEmpty(message = "{employee.lastName.required}")  private String lastName;    @NotEmpty(message = "{employee.designation.required}")  private String designation;    @NotNull(message = "{employee.salary.required}")  @Positive(message = "{employee.salary.invalid}")  private Integer salary;    @JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "yyyy-MM-dd")  @Past(message = "{employee.dateOfBirth.invalid}")  private Date dateOfBirth;  List<@NotEmpty(message = "{employee.email.invalid}") @Email(message = "{employee.email.invalid}") String> emails;           // setters and getters  } |

The @NotEmpty annotation validates that firstName, lastName, and designation fields are not empty. If they are, then an error message will be shown (more on this later). You may also use @Size annotation to validate the length of values of these fields.

@Past and @Positive annotations are self-explanatory. They will validate the corresponding field is in past date and positive number respectively.

A List of String stores all email addresses. As now we can declare validation constraints for the type of collections, you can see @NotEmpty and @Email are specified to validate emails as per requirement #4.

Common Validation Constraints

Following is a list of some common validations annotations:

|  |  |
| --- | --- |
| **Constraint** | **Description** |
| @Null | Annotated element must be null |
| @NotNull | Annotated element must not be null |
| @AssertTrue / @AssertFalse | Annotated element must not true or false respectively. Use with boolean type. |
| @Min | Annotated element must not a number greater or equal to specified minimum value. Example: @Min(value = 10) will accept values 10 and above. |
| @Positive / @Negative | Annotated element must be positive or negative number respectively. |
| @Digit | Annotated element must be a number within the accepted range. Specify the integral and fraction digits of a number. Example: @Digits(integer=10, fraction = 2) |
| @Size | Element’s size must be between given min/max values(included). |
| @Past / @Future | An instance, date or time must be in past or future resepectively. |
| @NotEmpty / @NotBlank | An element must not be empty or blank. |
| @Pattern | Annotated element (CharSequence) must match the specified regular expression. Example: @Pattern(regexp = “^[A-Za-z0-9]+$”) – to allow only alphanumeric characters. |
| @Email | A String with correct email address format. |

After annotating the bean you need to enable the validations where we accept the request bean(in Controller).

Enable Request Bean Validation

To enable the validation of request bean, just add @Valid along with @RequestBody of your request bean.



|  |  |
| --- | --- |
| 1  2  3  4 | @PostMapping  public ResponseEntity<EmployeeDO> addEmployee(@Valid @RequestBody EmployeeDO employeeDO) {  // code to add Employee  } |

This will call all validations in EmployeeDO. We need to build a response object which contains all validation error messages and give it to calling client. After getting our response object, calling client should do necessary actions(provide correct values). We will be handling this using @ControllerAdvice.

Handle Validation Exceptions

When validations of our annotated bean passed as method argument in Controller fails, it will throw MethodArgumentNotValidException. Spring already provides a class ResponseEntityExceptionHandler.java which handles this exception(along with some other exceptions) and returns a ResponseEntity.

We will be sending a custom response object ValidationErrorResponse.java which includes a string property of all error messages(comma separated) and detail of request for which validation fails.



|  |  |
| --- | --- |
| 1  2  3  4 | public class ValidationErrorResponse implements Serializable {  private String errorMessage;  private String requestDescription;  } |

To send above customize response when validation fails, we need to sub-class ResponseEntityExceptionHandler.java and override handleMethodArgumentNotValid() method.

CustomExceptionHandler.java



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | @ControllerAdvice  public class CustomExceptionHandler extends ResponseEntityExceptionHandler {    @Override  protected ResponseEntity<Object> handleMethodArgumentNotValid(MethodArgumentNotValidException ex,  HttpHeaders headers, HttpStatus status, WebRequest request) {  // Get the error messages for invalid fields  String errorMessage = ex.getBindingResult().getFieldErrors().stream()  .map(DefaultMessageSourceResolvable::getDefaultMessage).collect(Collectors.joining(", "));  // Create ValidationErrorResponse object using error messages and request details  ValidationErrorResponse errorResponse = new ValidationErrorResponse(errorMessage, request.getDescription(false));  return new ResponseEntity<Object>(errorResponse, HttpStatus.BAD\_REQUEST);  }  } |

Custom Validation Error messages

Though all validation annotations have their own default error messages, we always want to give a message which we feel as more explanatory for our client. Also, most of the time default error message doesn’t give complete information. For example, for not empty validation the default message is “must not be empty”. It doesn’t mention the name of the field. Also, sometimes its better to include the validated value in the error message.

All the annotations given above has an attribute “message”. Provide your custom message in this argument. But we don’t like to hard code the error messages, so we should externalize error messages to a properties file.

Hibernate Validator provides an implementation of [MessageInterpolator](https://docs.jboss.org/hibernate/beanvalidation/spec/2.0/api/javax/validation/MessageInterpolator.html) and loads all custom error messages from ValidationMessages.properties if available in the classpath. If you want to specify the validated value in the error message, just include ${validatedValue} in your custom error message.

All the validation annotations in our EmployeeDO has message attribute with a key of error message in ValidationMessages.properties. In our example it is:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | employee.firstName.required=First Name is required  employee.lastName.required=Last Name is required  employee.designation.required=Designation is required  employee.salary.required=Salary is required  employee.salary.invalid=Salary is invalid: '${validatedValue}'  employee.dateOfBirth.invalid=Date of Birth should be in past  employee.email.invalid=Invalid Email Address: '${validatedValue}' |

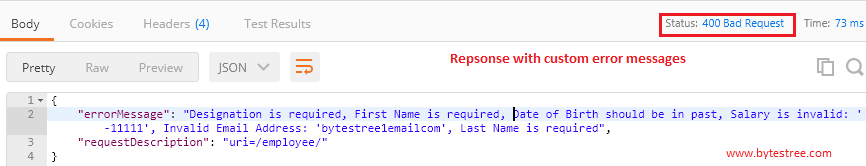
Demo

Let’s test our application using POSTMAN Rest client:

Provide all invalid values and see if we can get all custom error messages and request details in response.

**Request**:

**Response**:

[](http://www.bytestree.com/wp-content/uploads/2018/03/validation-CustomErrorMessages.png)

As you see all custom messages from  ValidationMessages.properties are shown in response.

And if you provide all correct values it will save an employee to the database. No screenshots for this, try it yourself :).