№6 Spring Data Validation

Spring Data Validation. Validation Rules. Error display. Custom Validators.

Java Bean Validation

https://beanvalidation.org/1.0/spec/
https://jcp.org/en/jsr/detail?id=303

Bean Validation 1.0 (2009) (JSR-303)
Bean Validation 1.1 (2013) (JSR-349)
Bean Validation 2.0 (2017)(JSR-330))

JSR | Community | Expert Group

Summary | Proposal | Detail (Summary & Proposal)

JSRs: Java Specification Requests

JSR 303: Bean Validation

Stage	Access	Start	Finish
Final Release	Download page	16 Nov, 2009	
Final Approval Ballot	View results	20 Oct, 2009	02 Nov, 2009
Proposed Final Draft	Download page	30 Mar, 2009	
Public Review Ballot	View results	03 Feb, 2009	09 Feb, 2009
Public Review	Download page	05 Jan, 2009	09 Feb, 2009
Early Draft Review	Download page	20 Mar, 2008	19 Apr, 2008
Expert Group Formation		25 Jul, 2006	21 Aug, 2007
JSR Review Ballot	View results	11 Jul, 2006	24 Jul, 2006

Status: Final
JCP version in use: 2.7
Java Specification Participation Agreement version in use: 2.0

Эталонной реализацией Bean Validation является Hibernate Validator. Bean Validation может использоваться не только в классических приложениях на основе Java EE, но и в приложениях на основе Spring, и даже в приложениях, не имеющих отношения к Java EE.

Для API Bean Validation потребуется зависимость **validation-api** из **javax.validation**:

```
<dependency>
    <groupId>javax.validation</groupId>
    <artifactId>validation-api</artifactId>
    <version>xxxxxx</version>
</dependency>
```

В Java есть интерфейс (javax.validation.Validator)

Валидация данных проводится в три базовых шага:

- Объявление «ограничений» (constraints) на данные в модели при помощи аннотаций
- Получение валидатора
- валидация объектов и обработка сообщений о нарушениях объявленных рамок (ConstraintViolations)

Используются готовые аннотации:
@Null, @DecimalMin, @Digits, Pattern, Email и др.

```
public class Person {

    @Size(min=2, max=50)
    private String Name;

    @Digits(integer=3, fraction=0, message = "Не более 3-х знаков")
    private Integer age;

    }
```

В Spring есть так же свой интерфейс Validator
(org.springframework.validation.Validator)

Validator можно использовать для проверки объектов. Интерфейс Validator работает с использованием объекта Errors, поэтому при проверке валидаторы могут сообщать об ошибках валидации объекту Errors.

```
public class Person {

    private String name;
    private int age;
      }
```

Реализация Validator довольно проста

```
public class PersonValidator implements Validator {

        public boolean supports(Class clazz) {
        return Person.class.equals(clazz);
    }

    public void validate(Object obj, Errors e) {
        ValidationUtils.rejectIfEmpty(e, "name", "name.empty");

         Person p = (Person) obj;
        if (p.getAge() < 0) {
            e.rejectValue("age", "negativevalue");
        } else if (p.getAge() > 110) {
            e.rejectValue("age", "too.darn.old");
        }
```

```
    }
}
```

Правила валидации

Правила валидации в Bean Validation задаются при помощи ограничений (constraints), аннотаций, расположенных в пакете **javax.validation.constraints**. Ограничения могут применяться к свойствам классов, аргументам методов и конструкторов, их возвращаемым значениям, а так же к типам обобщений.

Почитайте

https://beanvalidation.org/2.0/spec/

https://docs.jboss.org/hibernate/stable/validator/reference/en-US/html_single/

Числовые ограничения

Аннотация	Назначение
@DecimalMax	Применима к переменным типов **BigDecimal**, **BigInteger**, **CharSequence**, **byte**, **short** , **int**, **long** и их классов-обёрток. Значение должно быть меньше, либо равно указанному значению, либо быть **null** для непримитивов.
@DecimalMin	Аналогична @DecimalMax, значение переменной должно быть числом и быть больше, либо равной указанной значению, либо быть null для непримитивов.
@Digits	Количество символов слева от запятой должно быть меньше, либо равным **integer**, а справа — меньше, либо равным **fraction, null** является валидным значением. Применима к **BigDecimal**, **BigInteger**, **CharSequence**, **byte**, **short**, **int**, **long** и их классам-обёртам.
@Max	Значение должно быть меньше, либо равно указанному значению, либо быть **null**. Применима к переменным типов **BigDecimal**, **BigInteger**, **byte**, **short**, **int**, **long** и их классов-обёрток.
@Min	Значение должно быть больше, либо равно указанному значению, либо быть **null**. Применима к переменным типов **BigDecimal**, **BigInteger**, **byte**, **short**, **int**, **long** и их классов-обёрток.

@Negative	Значение должно быть отрицательным, либо быть **null**. Применима к переменным типов **BigDecimal**, **BigInteger**, **byte**, **short**, **int**, **long** и их классов-обёрток.
@NegativeOrZero	Значение должно быть отрицательным, равняться 0, либо быть **null**. Применима к переменным типов **BigDecimal**, **BigInteger**, **byte**, **short**, **int**, **long** и их классов-обёрток.
@Positive	Значение должно быть положительным, либо быть **null**. Применима к переменным типов **BigDecimal**, **BigInteger**, **byte**, **short**, **int**, **long** и их классов-обёрток.
@PositiveOrZero	Значение должно быть полоэительным, равняться 0, либо быть **null**. Применима к переменным типов **BigDecimal**, **BigInteger**, **byte**, **short**, **int**, **long** и их классов-обёрток.

Ограничения даты и времени

@Future	Значение переменной должно быть будущим временем. Применима к **Date**, **Calendar** и многим типам из пакета **java.time**.
@FutureOrPresent	Значение переменной должно быть будущим либо настоящим временем. Применима к **Date**, **Calendar** и многим типам из пакета **java.time**.
@Past	Значение переменной должно быть прошедшим временем. Применима к **Date**, **Calendar** и многим типам из пакета **java.time**.
@PastOrPresent	Значение переменной должно быть прошедшим либо настоящим временем. Применима к **Date**, **Calendar** и многим типам из пакета **java.time**.

Строчные ограничения

@Email	Значение должно быть адресом электронной почты; применима к **CharSequence**. Поведение зависит от конкретной реализации.
@NotBlank	Значение типа **CharSequence** не должно быть **null**, пустым или состоять из одних лишь пробельных символов.
@Pattern	Значение типа **CharSequence** должно соответствовать указанному регулярному выражению.

Булевые ограничения

@AssertFalse	Аннотация применима к переменным типов **boolean** и **Boolean**, значение которых должно быть **false**, либо **null**.
@AssertTrue	Противоположность @AssertFalse, значение должно быть **true** или **null**.

Универсальные

@NotEmpty	Значение типов **CharSequence**, **Collection**, **Map** или массив не должно быть **null** и должно содержать хотя бы 1 элемент.
@NotNull **@Null**	Значение не должно быть **null**. Значение должно быть **null**.
@Size	Размер значения типов **CharSequence, Collection, Map** или массива должен быть в указанном диапазоне — между **min** и **max.**

Hibernate Validation Constrains

https://docs.jboss.org/hibernate/validator/6.0/reference/en-US/pdf/hibernate_validator_reference.pdf

@ScriptAssert
Уровня класса проверяет класс на скрипты
@CreditCardNumber
Проверка на номер карты
@Currency
@DurationMax
@DurationMin
@ISBN
@EAN
@Length - Проверят длины строки –между минимальным и максимальным
@CodePointLength
@LuhnCheck
@Mod10Check
@Mod11Check
@Range
@SafeHtml
@UniqueElements – содержит ли коллекция уникальные элементы
@Url – проверка на валидацию URL

Пример:

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class PersonForm {

    @NotNull(message="Name is required")
    @Size(min=3, message="Name must be at least 3 characters long")
    private String firstName;
    @NotBlank(message="LastName is required")
    private String lastName;
    @NotBlank(message="Street is required")
    private String street;
    @NotBlank(message="City is required")
    private String city;
    @Digits(integer=6, fraction=0, message="Invalid zip cpde")
    private String zip;
    @NotBlank(message="Email is required")
    @Email (message = "Enter the email in correct format")
    private String email;

  //  @Pattern(regexp="^(0?[1-9]|[12][0-9]|3[01])[\\/\\-](0?[1-9]|1[012])[\\/\\-]\\d{4}$",message="Must be formatted DD/MM/YYYY")
  //  private String birthday;
    //ISO 8601 date format (yyyy-MM-dd)
    @DateTimeFormat(iso = DateTimeFormat.ISO.DATE)
```

```
    @Past(message = "Date is not valid")
    private Date birthday;

    private String phone;
}
```

Проверка

Чтобы запустить проверку ввода @Controller, просто аннотируйте входной аргумент как @Valid:

```
@Controller
public class MyController {

    @RequestMapping("/foo", method=RequestMethod.POST)
    public void processFoo(@Valid Foo foo) { /* ... */ }
```

Spring MVC будет проверять объект @Valid после привязки, когда будет настроен соответствующий Validator.

Для этого добавляем аннотацию **@Valid** к аргументу метода:

```
@PostMapping // POST - create new
public ResponseEntity handlePost (@Valid  @RequestBody CompyDto
compyDto){

    CompyDto savedDto =  compyService.savedCompy(compyDto);

    HttpHeaders headers = new HttpHeaders();
    headers.add("Location", "/api/v1/compy/" +
savedDto.getId().toString());
    return  new ResponseEntity(headers,HttpStatus.CREATED);

   }
```

Аннотация *@Valid* указывает Spring MVC выполнить валидацию отправленного объекта *CompyDto* после его привязки к отправленным данным и до вызова метода `handlePost` (). Если есть ошибки валидации, детали этих ошибок будут записаны в объекте Errors, который также передается в метод.

https://docs.oracle.com/javaee/7/tutorial/partbeanvalidation.htm#sthref1322

Validation Error Handling

Генерация исключения из любого веб-уровня гарантирует, что Spring отобразит соответствующий код состояния в ответе HTTP:

Другой вариант сопоставления пользовательских исключений с конкретными кодами состояния - использовать аннотацию **@ExceptionHandler** в контроллере. Проблема этого подхода заключается в том, что аннотация применяется только к контроллеру, в котором он

определен. Это означает, что нужно объявлять в каждом контроллере индивидуально.

В котроллер добавим метод

```
@ExceptionHandler(ConstraintViolationException.class)
public ResponseEntity<List> validationErrorHandler (ConstraintViolationException e){
    List<String> errors =  new ArrayList<>(e.getConstraintViolations().size());
    e.getConstraintViolations().forEach(constraintViolation ->
            errors.add(constraintViolation.getPropertyPath() + " : " +
constraintViolation.getMessage()));
    return new ResponseEntity<>(errors, HttpStatus.BAD_REQUEST);
    }
}
```

- Аннотация ExceptionHandler. - для обработки собственных и каких-то специфичных исключений. https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/ExceptionHandler.html
- Аннотация ControllerAdvice. Данная аннотация дает «совет» группе контроллеров по определенным событиям. В нашем случае — это обработка ошибок. По умолчанию применяется ко всем контроллерам, но в параметрах можно указать отпределенную группу.

https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/ControllerAdvice.html

Добавим еще один метод

```
@ExceptionHandler(BindException.class)
public ResponseEntity<List> handleBindException (BindException ex){
    return new ResponseEntity(ex.getMessage(), HttpStatus.BAD_REQUEST);
}
```

Можно написать такой контроллер

```
import by.patsei.springproject1.exceptions.ResourceNotFoundException;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.ResponseStatus;

@ControllerAdvice
public class ResourceNotFound {

        @ResponseBody
        @ExceptionHandler(ResourceNotFoundException.class)
        @ResponseStatus(HttpStatus.NOT_FOUND)
        String recourceNotFoundHandler  (ResourceNotFoundException ex) {
            return ex.getMessage();
        }
    }
```

Пользовательские валидаторы

Напишем пользовательский валидатор.

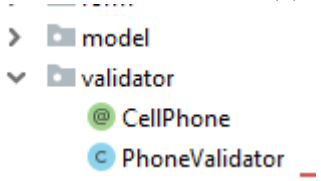
Для этого добавим новое поле – phone – телефонный номер, которое будем валидировать.

Person.class

```
@Data
@AllArgsConstructor
public class Person {
    private String firstName;
    private String lastName;
    private String street;
    private String city;
    private String zip;
    private String email;
    private Date birthday;
    private String phone;
}
```

В Spring есть свой интерфейс Validator. (org.springframework.validation.Validator) как и в Java (javax.validation.Validator). Его имплементация выполняет проверку данных. Таким образом, есть две возможности - использовать спецификацию JSR-303 и его класс Validator или сделать реализацию интерфейса org.springframework.validation.Validator.

Пользовательские валидаторы разместим в отдельном пакете.



Для создания пользовательской аннотации ее надо создать. Добавьте новую аннотацию CellPhone. Он будет проверять формат телефона

```
@Documented
@Constraint(validatedBy = PhoneValidator.class)
@Target ({ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface CellPhone {

    String message() default "{Phone}";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

Большая часть определения аннотации стандартна и соответствует спецификации JSR-303.

@Target – указывает, что именно мы можем пометить этой аннотацией.
ElementType.PACKAGE – только для пакетов;
ElementType.TYPE – только для классов;
ElementType.CONSTRUCTOR – только для конструкторов;
ElementType.METHOD – только для методов;
ElementType.FIELD – только для атрибутов(*переменных*) класса;
ElementType.PARAMATER – только для параметров метода;
ElementType.LOCAL_VARIABLE – только для локальных переменных.

`@Target ({ElementType.FIELD})` - сообщает о там, что аннотация будет применяться к полю.

`@Documented` – указывает, что помеченная таким образом аннотация должна быть добавлена в javadoc поля/метода.
@Retention – позволяет указать жизненный цикл аннотации: будет она присутствовать только в исходном коде, в скомпилированном файле, или она будет также видна и в процессе выполнения.
RetentionPolicy.CLASS – будет присутствовать в скомпилированном файле;
RetentionPolicy.RUNTIME – будет присутствовать только в момент выполнения;
RetentionPolicy.SOURCE – будет присутствовать только в исходном коде.

Наиболее важной частью является аннотация @**Constraint**, где мы предоставляем класс, который будет использоваться для проверки, т.е. *PhoneValidator*.

Реализация класса проверки выполняется в классе *PhoneValidator.class*

```
public class PhoneValidator implements ConstraintValidator<CellPhone,
String> {

    @Override
    public void initialize(CellPhone paramA) {
    }
    @Override
    public boolean isValid(String phoneNo, ConstraintValidatorContext
ctx) {
        if(phoneNo == null){
            return false;
        }
        //задание номера телефона в формате  "123456789"
        if (phoneNo.matches("\\d{9}"))
                    return true;
            //номер телефона может разделяться -, . или пробелом
        else if(phoneNo.matches("\\d{2}[-\\.\\s]\\d{3}[-
\\.\\s]\\d{2}[-\\.\\s]\\d{2}"))
```

```
                    return true;
                //может быть код оператора в скобках ()
            else if(phoneNo.matches("\\(\\d{2}\\)\\d{3}\\d{2}\\d{2}"))
                    return true;
                //может быть код страны в скобках ()
            else if(phoneNo.matches("\\(\\d{3}\\)\\d{2}\\d{3}\\d{4}"))
                    return true;
                //return false если ничего не подходит
            else return false;
        }

}
```

Необходимо чтобы класс реализовал интерфейс javax.validation.ConstraintValidator.

Если используются ресурс, например DataSource, то можно инициализировать его в методе initialize ().

Основную логику проверки выполняет метод **isValid(String phoneField, ConstraintValidatorContext cxt)**. Значение поля передается в качестве первого аргумента Метод проверки isValid возвращает true, если данные верны, иначе - false.

Проверка выполняется на основе регулярного выражения.

Аннотация готова, добавляем ее к полю и уже можно проверить, все поля на которых есть аннотации будут проверены соответствующими правилами.

Сообщения об ошибках можно вынести в отдельный файл ресурсов. Это удобно, если делать локализацию на нескольких языках (русский+ английский).

Создадим файл ресурсов

validationmessages.properties

```
#message for user validator
Phone = "Phone is not valid"
valid.phone.cellphone = "Phone format is not valid"

valid.name.notNull = "Name is required"
valid.firstname.size.min3 = "Name must be at least 3 characters
long"
valid.lastname.notBlank="LastName is required"
valid.street.notBlank = "Street is required"
valid.city.notBlank = "City is required"
valid.zip.digits = "Invalid zip cpde"
valid.email.notBlank = "Email is required"
valid.email.email = "Enter the email in correct format"
valid.birthday.past = "Date is not valid"
```

Добавим опеределение LocalValidatorFactoryBean в конфигурации приложения. Ниже показан пример.

```
@SpringBootApplication
public class SpringProject1Application {

   @Bean
   public MessageSource messageSource() {
      ReloadableResourceBundleMessageSource messageSource = new
ReloadableResourceBundleMessageSource();
      messageSource.setBasename("classpath:validationmessages");
      messageSource.setDefaultEncoding("UTF-8");
      return messageSource;
   }
   @Bean
   public LocalValidatorFactoryBean validator() {
      LocalValidatorFactoryBean bean = new
LocalValidatorFactoryBean();
      bean.setValidationMessageSource(messageSource());
      return bean;
   }

   public static void main(String[] args) {

       SpringApplication.run(SpringProject1Application.class,
args);
   }
}
```

Изменим класс PersonForm

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class PersonForm {

    @NotNull(message="{valid.name.notNull}")
    @Size(min=3, message="{valid.firstname.size.min3}")
    private String firstName;
    @NotBlank(message="{valid.lastname.notBlank}")
    private String lastName;
    @NotBlank(message="{valid.street.notBlank}")
    private String street;
    @NotBlank(message="{valid.city.notBlank}")
    private String city;
    @Digits(integer=6, fraction=0, message="{valid.zip.digits}")
    private String zip;
    @NotBlank(message="{valid.email.notBlank}")
    @Email (message = "{valid.email.email}")
    private String email;

  //  @Pattern(regexp="^(0?[1-9]|[12][0-9]|3[01])[\\/\\-](0?[1-
9]|1[012])[\\/\\-]\\d{4}$",message="Must be formatted
DD/MM/YYYY")
  //  private String birthday;

    //ISO 8601 date format (yyyy-MM-dd)
```

```
    @DateTimeFormat(iso = DateTimeFormat.ISO.DATE)
    @Past(message = "{valid.birthday.past}")
    private Date birthday;


    @CellPhone (message = "{valid.phone.cellphone}")
    private String phone;

}
```