

Тема 4 Spring Framework Validation

Spring поддерживает Java Bean Validation API (также известный как JSR-303 первая версия – это часть Java EE, текущая версия — 2.0, является частью Java EE 8 и описана в JSR-380), что позволяет объявлять правила проверки.

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.

Со Spring Boot не нужно делать ничего для добавления библиотек валидации, потому что API валидации и реализация Hibernate автоматически добавляются в проект как временные зависимости веб-стартера Spring Boot.

В принципе для API Bean Validation потребуется зависимость **validation-api** из **javax.validation**:

```
<dependency>
2   <groupId>javax.validation</groupId>
3   <artifactId>validation-api</artifactId>
4   <version>xxxxxx</version>
5 </dependency>
```

В Spring есть так же свой интерфейс Validator

(org.springframework.validation.Validator)

Чтобы применить проверку необходимо:

- a) объявить правила проверки для класса,
- b) указать, что валидация должна выполняться в методах контроллера, которые требуют валидации.
- c) изменить виды формы для отображения ошибок проверки.

4.1 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.

Для проверки объекта используются аннотации на полях класса, т.е. декларативная модель. Уже готовые: @Null, @DecimalMin, @Digits, Pattern, Email и др. А также можно делать и собственные.

Почитайте

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

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

4.1.1 Числовые ограничения

@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-обёрткам.

@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 положительным, равняться 0, либо быть **null**.
Применима к переменным типов **BigDecimal, BigInteger, byte, short, int, long** и их классов-обёрток.

4.1.2 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**.

4.1.3 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.

4.1.4 Булевыe 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**.

4.1.5 Универсальные

@NotEmpty

Значение типов **CharSequence**, **Collection**, **Map** или массив не должно быть **null** и должно содержать хотя бы 1 элемент.

@NotNull

Значение не должно быть **null**.

@Null

Значение должно быть **null**.

@Size

Размер значения типов **CharSequence**, **Collection**, **Map** или массива должен быть в указанном диапазоне — между **min** и **max**.

4.1.6 Пример

Добавим требования:

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

    @NotBlank(message="Name is required")
    @Size(min=3, message="Name must be at least 3 characters long")
    private String firstName;
```

Здесь мы определяем, что свойство `firstName` должно иметь значение длиной не менее 3 символов. Практически для всех полей надо поставить требование, чтобы пользователь не оставил ни одно из полей пустым. Для можно использовать аннотацию Hibernate Validator `@NotBlank`.

Однако проверка поля индекса отличается. Вам нужно не только убедиться, что свойство не пустое, но и содержать числовое значение не менее 6 символов. Поэтому

```
@Digits(integer=6, fraction=0, message="Invalid zip code")
private String zip;
```

Значения можно проверять и так:

```
@NotNull
@Min(18)
private Integer age;
```

Те. Значения не может быть меньше 18.

Для проверки значения по шаблону можно использовать регулярные выражения:

```
@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;
```

Есть еще много полезных аннотаций валидации. Для валюты:

```
@NumberFormat(style= NumberFormat.Style.CURRENCY)
```

Если нужно сделать проверку параметров

```
public void createUser(@Email String email,
```

```

        @NotNull String name) {
    ...
}

```

Все аннотации валидации включают атрибут `message`, который определяет сообщение для пользователя, если введенная информация не соответствует требованиям правил проверки.

4.2 Проверка

Теперь, когда вы объявили, как должен проверяться объект, нужно вернуться к контроллеру и указать на необходимость проверки, когда форма отправляется в метод-обработчик.

Это делается путем размещения аннотации `@Valid` непосредственно перед аннотацией `@RequestBody` в контроллере.

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

```

@PostMapping(value = {"/addPerson"})
public ModelAndView savePerson( Model model,
                               @Valid @ModelAttribute("personForm") PersonForm
                               personForm,
                               Errors errors) {
    ModelAndView modelAndView = new ModelAndView();
    if (errors.hasErrors()) {
        modelAndView.setViewName("addPerson");
    }
    else {
        modelAndView.setViewName("personList");
        String firstName = personForm.getFirstName();
        String lastName = personForm.getLastName();
        String street = personForm.getStreet();
        String city = personForm.getCity();
        String zip = personForm.getZip();
        String email = personForm.getEmail();
        String birthday = personForm.getBirthday();
        Person newPerson = new Person(firstName, lastName, street,
                                       city, zip, email, birthday);
        persons.add(newPerson);
        model.addAttribute("persons", persons);
        log.info("/addPerson - POST was called");
        return modelAndView;
    }
    return modelAndView;
}

```

Аннотация `@Valid` указывает Spring MVC выполнить валидацию отправленного объекта `PersonForm` после его привязки к отправленным данным формы и до вызова метода `savePerson()`. Если есть ошибки валидации, детали этих ошибок будут записаны в объекте `Errors`, который также передается в метод. В начале метода обращаемся к объекту `Errors`, вызываем

его метод `hasErrors()` - если есть какие-либо ошибки проверки? Если есть - возвращаем имя представления «`addPerson`», чтобы форма снова отображалась и завершаем метод.

По ссылке можно получить официальную информацию по валидации <https://docs.oracle.com/javaee/7/tutorial/partbeanvalidation.htm#sthref1322>

4.3. Отображение ошибок

Spring предлагает несколько шаблонов для определения представлений: JavaServer Pages (JSP), Thymeleaf, FreeMarker, Mustache и Groovy. В данном проекте мы использовали Thymeleaf.

Шаблоны Thymeleaf - это просто HTML с некоторыми дополнительными атрибутами элемента. Например, если вы хотите отображение информации :

```
<th>Last Name</th>
```

или

```
<h2 th:utext="${message}">...!..</h2>
```

В последнем случае, когда шаблон отображается в HTML, тело элемента будет заменено значением атрибута запроса сервлета, ключ которого - «`message`».

Атрибут `th: utext` является атрибутом Thymeleaf-namespaced, который выполняет замену.

Оператор `$ {}` говорит ему использовать значение атрибута запроса. Thymeleaf также предлагает другой атрибут `th: each`, который выполняет итерацию по коллекции элементов, отображая HTML один раз для каждого элемента в коллекции. Это используется в перечислении. Например, чтобы отобразить только список:

```
<tr th:each ="person : ${persons}">
  <td th:utext="${person.firstName}">...</td>
  <td th:utext="${person.lastName}">...</td>
</tr>
```

Thymeleaf предлагает удобный доступ к объекту `Errors` через свойство `fields` и его атрибут `th: errors`. Например, чтобы отобразить ошибки валидации можно добавить элемент `` в `addPerson.html`, который использует эти ссылки на ошибки:

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
  <meta charset="UTF-8" />
  <title>Add Person</title>
```

```

<link rel="stylesheet" type="text/css"
th:href="@{/css/style.css}"/>
</head>
<body>
<h1>Create a Person:</h1>

<form th:action="@{/addPerson}"
      th:object="${personForm}" method="POST">

    <div th:if="${#fields.hasErrors()}">
    <span class="validationError">
    Please correct the information.
    </span>
    </div>

    First Name:
    <input type="text" th:field="*{firstName}" />
    <span class="validationError"
          th:if="${#fields.hasErrors('firstName')}"
          th:errors="*{firstName}">
    </span>
    <br/>

```

В стиль можно добавить чтобы эти сообщения выводились курсивом и красным цветом. Вставьте вывод ошибок для остальных поле ввода. Запустите приложение и проверьте как работает валидация:

Create a Person:

Please correct the information.

First Name: *Name must be at least 3 characters long*

Last Name:

Street address:

City:

Zip code:

Email:

Birthday: *Must be formatted DD/MM/YYYY*

4.4 Пользовательские валидаторы

Большую часть проверок можно сделать встроенными аннотациями, но иногда необходимо сделать проверку уникальности отдельных значений (паспорта, телефона, УНН, карты и т.д.). В таком случае существующими стандартными аннотациями не обойдётся, более того, иногда нужно обращаться к базе данных для проверки существования записи. В таком случае нужно писать свою аннотацию.

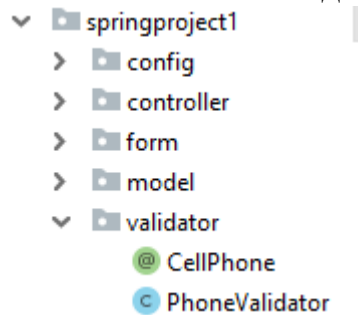
Напишем пользовательский валидатор.

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. Это уже не декларативный подход, но в нем есть своя гибкость и расширяемость. 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.

`String message()` — здесь мы указываем сообщение об ошибке по умолчанию. `groups()` и `payload()` в данном случае не пригодятся, поэтому они пустые.

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`. 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 ctx)`**. Значение поля передается в качестве первого аргумента. Метод проверки `isValid` возвращает `true`, если данные верны, иначе - `false`.

Проверка выполняется на основе регулярного выражения.

```
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}[-\\.\\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;
    }
}
```

```

    }
}

```

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

```

@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;

    @CellPhone (message = "Phone format is not valid")
    private String phone;
}

```

Проверка выполняется при добавлении нового объекта. Поэтому изменим метод

```

@PostMapping(value = {"/addPerson"})
public ModelAndView savePerson( Model model, //
                                @Valid @ModelAttribute("personForm")
                                PersonForm personForm, Errors errors) {

```

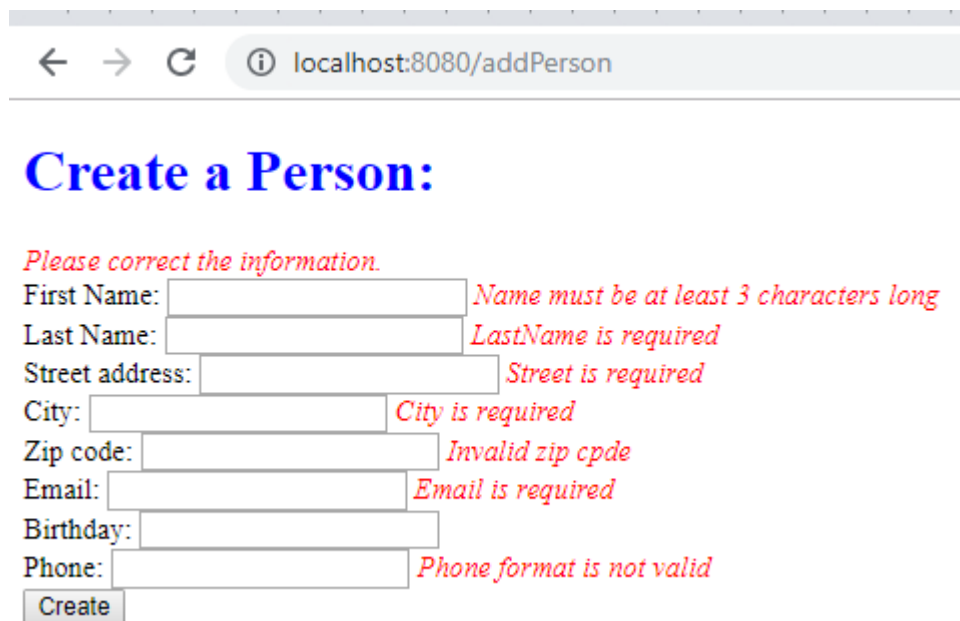
```

ModelAndView modelAndView = new ModelAndView();
if (errors.hasErrors()) {
    modelAndView.setViewName("addPerson");
}
else {
    modelAndView.setViewName("personList");
    String firstName = personForm.getFirstName();
    String lastName = personForm.getLastName();
    String street = personForm.getStreet();
    String city = personForm.getCity();
    String zip = personForm.getZip();
    String email = personForm.getEmail();
    Date birthday = (Date)personForm.getBirthday();
    String phone = personForm.getPhone();

    Person newPerson = new Person(firstName, lastName, street, city, zip,
email, birthday, phone);
    persons.add(newPerson);
    model.addAttribute("persons", persons);
    log.info("/addPerson - POST was called");
    return modelAndView;
}
return modelAndView;
}

```

Тестируем приложение

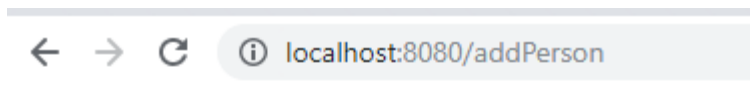


The screenshot shows a web browser window with the address bar displaying 'localhost:8080/addPerson'. The page title is 'Create a Person:'. Below the title, there is a red message: 'Please correct the information.' followed by several input fields with associated error messages:

- First Name: Name must be at least 3 characters long
- Last Name: LastName is required
- Street address: Street is required
- City: City is required
- Zip code: Invalid zip cpde
- Email: Email is required
- Birthday:
- Phone: Phone format is not valid

At the bottom of the form is a 'Create' button.

Теперь введем верный формат телефона (согласно коду у нас есть несколько вариантов)



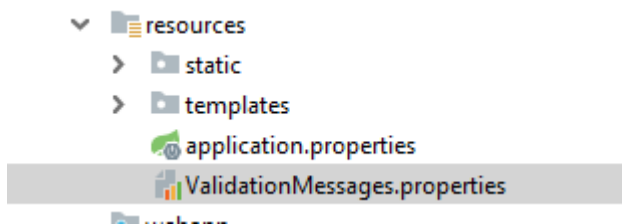
Create a Person:

Please correct the information.

First Name:	<input type="text" value="Natallia"/>
Last Name:	<input type="text" value="Patsei"/>
Street address:	<input type="text" value="Sverdlova 23"/>
City:	<input type="text" value="Minsk"/>
Zip code:	<input type="text" value="220123"/>
Email:	<input type="text" value="n.patsei@yy.by"/>
Birthday:	<input type="text" value="2020-12-01"/> <i>Date is not valid</i>
Phone:	<input type="text" value="123456789"/>
<input type="button" value="Create"/>	

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

Создадим файл ресурсов



Со следующим содержимым:

```
#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 определение 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);
    }

}
```

ReloadableResourceBundleMessageSource является наиболее распространенной реализацией MessageSource, которая разрешает сообщения из комплектов ресурсов для разных локалей.

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(regex="^(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;

}
```