Основы валидации Java Bean

Все(170 листов) - <https://docs.jboss.org/hibernate/stable/validator/reference/en-US/html_single/#validator-annotationprocessor-options>

**Валидация это наподобие “пробрасываемых наверх” исключений *(типа я тут вам “просимофорил”, что это не правильно, а вы сами дальше решайте, как поступать с нарушением данного условия)*.**

**Нельзя применять к статическим полям и методам!!!!**

**1. Обзор**

Для валидации данных в Java EE существует [Bean Validation](https://beanvalidation.org/" \t "_blank). Первая версия данного набора API была специфицирована в [JSR-303](https://www.jcp.org/en/jsr/detail?id=303) и опубликована как часть Java EE 6. Текущая версия — 2.0, является частью Java EE 8 и описана в [JSR-380](https://www.jcp.org/en/jsr/detail?id=380). Эталонной реализацией Bean Validation является [Hibernate Validator](http://hibernate.org/validator/" \t "_blank). 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.

Конечно, проверка пользовательского ввода является супер распространенным требованием в большинстве приложений, и инфраструктура Java Bean Validation стала стандартом де-факто для обработки логики такого рода.

**2. JSR 380**

JSR 380 - это спецификация Java API для проверки компонентов, часть JavaEE и JavaSE, которая обеспечивает соответствие свойств компонента определенным критериям с использованием аннотаций, таких как *@ NotNull*, *@ Min*и *@ Max*.

Эта версия требует Java 8 или выше, и использует преимущества новых функций, добавленных в Java 8, таких как аннотации типов, и поддерживает новые типы, такие как *Optional*и *LocalDate*.

Чтобы получить полную информацию о спецификациях, прочитайте [JSR 380](https://jcp.org/en/jsr/detail?id=380).

**3. зависимости**

Мы собираемся использовать пример Maven, чтобы показать необходимые зависимости, но, конечно, эти файлы могут быть добавлены различными способами.

**3.1. Реализация API-интерфейса валидации**

* Hibernate Validator является эталонной реализацией API валидации. \*\*

Чтобы использовать его, мы должны добавить следующие зависимости.

Начиная с Hibernate Validator 6.1.5.Final идет поддержка Java 9

<dependency>

<groupId>org.hibernate.validator</groupId>

<artifactId>hibernate-validator</artifactId>

<version>6.1.5.Final </version>

</dependency>

Это транзитивно вытягивает зависимость от API Jakarta Bean Validation ( jakarta.validation:jakarta.validation-api:2.0.2).

*Следующая настройка подключает блок, который следит за правильностью мест, где будут установлены аннотации валидации(также этот блок можно устанавливать в настройках большинства самих IDE):*

<dependency>

<groupId>org.hibernate.validator</groupId>

<artifactId>hibernate-validator-annotation-processor</artifactId>

<version>6.1.5.Final</version>

</dependency>

Вкратце отметим, что ***hibernate-validator* полностью отделен от аспектов персистентности Hibernate** и, добавляя его в качестве зависимости, мы не добавляем эти аспекты персистентности в проект.

Начиная с версии 1.1, Bean Validation (и, следовательно, Jakarta Bean Validation) интегрирован с CDI (контексты и внедрение зависимостей для Jakarta EE).

Эта интеграция обеспечивает управляемые CDI компоненты для Validator и ValidatorFactory и позволяет вставлять зависимости в валидаторы ограничений, а также пользовательские интерполяторы сообщения.

Если ваше приложение выполняется в контейнере Java EE, эта интеграция включена по умолчанию. При работе с CDI в контейнере сервлетов или в чистой среде Java SE вы можете использовать портативное расширение CDI, предоставляемое Hibernate Validator. Для этого добавьте переносимое расширение в ваш путь к классу:

*Пример 1.3: Переносимое расширение CDI Hibernate Validator Зависимость Maven*

<dependency>

<groupId>org.hibernate.validator</groupId>

<artifactId>hibernate-validator-cdi</artifactId>

<version>6.1.5.Final</version>

</dependency>

**3.2. Зависимости языка выражений**

В документации docs.jboss.org написано следующее:

Hibernate Validator требует реализации языка [выражений Jakarta](https://projects.eclipse.org/projects/ee4j.el) для оценки динамических выражений в сообщениях о нарушении ограничений. Если ваше приложение выполняется в контейнере Java EE, таком как JBoss AS, реализация EL уже предусмотрена контейнером. Однако **в среде Java SE** вы **должны добавить реализацию** в качестве зависимости **к** вашему **POM-файлу**. Например, вы можете добавить следующую зависимость для использования [эталонной реализации](https://github.com/eclipse-ee4j/el-ri) Jakarta EL :

*Пример 1.2: зависимости Maven для эталонной реализации EL*

<dependency>

<groupId>org.glassfish</groupId>

<artifactId>jakarta.el</artifactId>

<version>3.0.3</version>

</dependency>

А автор этой статьи пишет следующее:

JSR 380 обеспечивает поддержку интерполяции переменных, что позволяет использовать выражения внутри сообщений нарушения.

Чтобы разобрать эти выражения, мы должны добавить зависимость как от API языка выражений, так и от реализации этого API. GlassFish обеспечивает эталонную реализацию:

<dependency>

<groupId>javax.el</groupId>

<artifactId>javax.el-api</artifactId>

<version>3.0.0</version>

</dependency>

<dependency>

<groupId>org.glassfish.web</groupId>

<artifactId>javax.el</artifactId>

<version>2.2.6</version>

</dependency>

Если эти JAR-файлы не добавлены, вы получите сообщение об ошибке во время выполнения, как показано ниже:

*HV000183: Невозможно загрузить «javax.el.ExpressionFactory». Убедитесь, что у вас есть зависимости EL от пути к классам, или используйте ParameterMessageInterpolator вместо*

**4. Использование аннотаций валидации**

Аннотаций валидации можно устанавливать на:

- переменных,

- элементах коллекций,

- методах,

- классах.

В качестве основного примера мы будем использовать bean-компонент *User*и поработаем над добавлением к нему простой проверки:

import javax.validation.constraints.AssertTrue;

import javax.validation.constraints.Max;

import javax.validation.constraints.Min;

import javax.validation.constraints.NotNull;

import javax.validation.constraints.Size;

import javax.validation.constraints.Email;

public class User {

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

private String name;

@AssertTrue

private boolean working;

@Size(min = 10, max = 200, message

= "About Me must be between 10 and 200 characters")

private String aboutMe;

@Min(value = 18, message = "Age should not be less than 18")

@Max(value = 150, message = "Age should not be greater than 150")

private int age;

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

private String email;

//standard setters and getters

}

Все аннотации, используемые в примере, являются стандартными аннотациями JSR:

***@ NotNull***- подтверждает, что аннотированное значение свойства не является *ноль*

***@ AssertTrue***\*\* - проверяет, что аннотированное значение свойства *правда*

***@ Size***\*\* - подтверждает, что аннотированное значение свойства имеет размер между атрибутами *min*и *max*; может применяться к *String*, *Collection*, *Map*и свойства массива

***@ Min***\*\* - *\_v \_*Проверяет, что аннотированное свойство имеет значение не меньше, чем атрибут *value*

***@ Max***\*\* - проверяет, что аннотированное свойство имеет значение не больше чем атрибут *value*

**@ Email** \*\* - подтверждает, что аннотированное свойство является действительным адресом электронной почты адрес

Некоторые аннотации принимают дополнительные атрибуты, но атрибут *message*является общим для всех них. Это сообщение, которое обычно отображается, когда значение соответствующего свойства не проходит проверку.

Некоторые дополнительные аннотации, которые можно найти в JSR:

***@ NotEmpty***- проверяет, что свойство не является нулевым или пустым; Может быть примененным к значениям *String*, *Collection*, *Map*или *Array*

**@ NotBlank** \*\* - может применяться только к текстовым значениям и подтверждено, что свойство не является пустым или пустым

**@ Positive** и**@ PositiveOrZero** \*\* - применяются к числовым значениям и подтвердить, что они строго положительные или положительные, включая 0

**@ Negative** и**@ NegativeOrZero** \*\* - применяются к числовым значениям и подтвердить, что они строго отрицательные или отрицательные, включая 0

**@ Past** и**@ PastOrPresent** \*\* - проверить, что значение даты находится в прошлом или прошлом, включая настоящее; может применяться к типам дат в том числе добавленные в Java 8

**@ Future** и**@ FutureOrPresent** \*\* - проверяет, что значение даты находится в будущем или будущем, включая настоящее

* Проверочные аннотации также могут применяться к элементам коллекции \*\* :

List<@NotBlank String> preferences;

В этом случае любое значение, добавленное в список настроек, будет проверено.

**@NotBlank** применяется только к строкам и проверяет, что строка не пуста.  
**@NotNull** применяется к CharSequence, Collection, Map или Array и проверяет, что объект не равен null. Но при этом он может быть пуст.  
**@NotEmpty** применяется к CharSequence, Collection, Map или Array и проверяет, что он не null имеет размер больше 0.

Аннотация **@Size**(min=6) пропустит строку состоящую из 6 пробелов и/или символов переноса строки, а **@NotBlank** не пропустит.

Спецификация **также поддерживает новый тип *Optional***в Java 8:

private LocalDate dateOfBirth;

public Optional<@Past LocalDate> getDateOfBirth() {

return Optional.of(dateOfBirth);

}

Здесь платформа проверки автоматически развернет значение *LocalDate*и проверит его.

**5. Проверка установленных аннотаций**

Некоторые фреймворки, такие как Spring, имеют простые способы запуска процесса проверки, просто используя аннотации. Это главным образом для того, чтобы нам не приходилось взаимодействовать с программным API проверки.

Давайте теперь пойдем по ручному маршруту и ​​настроим все программно:

ValidatorFactory factory = Validation.buildDefaultValidatorFactory();

Validator validator = factory.getValidator();

Чтобы проверить бин, мы должны сначала иметь объект *Validator*, который создается с использованием *ValidatorFactory*.

**5.1. Определение bean**

Теперь мы собираемся создать пользователя User (из главы 4) с недопустимым нулевым значением *name*:

User user = new User();

user.setWorking(true);

user.setAboutMe("Its all about me!");

user.setAge(50);

**5.2. Подтвердите bean**

Теперь, когда у нас есть *Validator*, мы можем проверить наш компонент, передав его методу *validate*. Любые нарушения ограничений, определенных в объекте *User*, будут возвращены как *Set*.

Set<ConstraintViolation<User>> violations = validator.validate(user);

Итерируя нарушения, мы можем получить все сообщения о нарушениях, используя метод *getMessage*.

for (ConstraintViolation<User> violation : violations) {

log.error(violation.getMessage());

}

В нашем примере ( *ifNameIsNull*nameValidationFails *) набор будет содержать одну*ConstraintViolation *с сообщением «*Name cannot be null ».

**6. Валидация методов**

*Аннотации ограничений к параметрам метода и конструктора*

package org.hibernate.validator.referenceguide.chapter03.parameter;

public class RentalStation {

public RentalStation(@NotNull String name) {

//...

}

public void rentCar(

@NotNull Customer customer,

@NotNull @Future Date startDate,

@Min(1) int durationInDays) {

//...

}

}

*Аннотации ограничений к параметрам или возвращаемому значению исполняемого метода (чаще всего используются с самостоятельно созданными аннотациями)*

package org.hibernate.validator.referenceguide.chapter03.crossparameter.constrainttarget;

public class Garage {

@ELAssert(expression = "...", validationAppliesTo = ConstraintTarget.PARAMETERS)

public Car buildCar(List<Part> parts) {

//...

return null;

}

@ELAssert(expression = "...", validationAppliesTo = ConstraintTarget.RETURN\_VALUE)

public Car paintCar(int color) {

//...

return null;

}

}

, где:

- ConstraintTarget.PARAMETERS – указывает на то, что эта аннотация применяется для ограничения параметров метода,

- ConstraintTarget.RETURN\_VALUE – указывает на то, что эта аннотация применяется для ограничения возвращаемого значения метода

**7. При аннотировании элементов с “ленивой” загрузкой (например в Hibernate или JPA)**

Проверка этого ленивого свойства или ассоциации будет означать, что к его состоянию необходимо будет получить доступ, инициируя загрузку из базы данных.

Какие свойства могут быть доступны, а какие нет, управляется запросом интерфейса TraversableResolver:

package org.hibernate.validator.referenceguide.chapter09;

Создаем свой класс, где будут указаны разрешенные к моментальной загрузке (для проверки) элементы (не указанные элементы будут запрещены):

public class MyTraversableResolver implements TraversableResolver {

@Override

public boolean isReachable(

Object traversableObject,

Node traversableProperty,

Class<?> rootBeanType,

Path pathToTraversableObject,

ElementType elementType) {

//...

return false;

}

@Override

public boolean isCascadable(

Object traversableObject,

Node traversableProperty,

Class<?> rootBeanType,

Path pathToTraversableObject,

ElementType elementType) {

//...

return false;

}

}

После чего создаем валидатор, в котором применяем наши ограничения:

ValidatorFactory validatorFactory = Validation.byDefaultProvider()

.configure()

**.traversableResolver( new MyTraversableResolver() )**

.buildValidatorFactory();

Validator validator = validatorFactory.getValidator();

По умолчанию отслеживаемые вызовы распознавателя кэшируются для каждого вызова проверки. Это особенно важно в среде JPA, где это требует значительных затрат.

Это кэширование добавляет некоторые накладные расходы. Возможно “кеш” лучше отключить.

ValidatorFactory validatorFactory = Validation.byProvider( HibernateValidator.class )

.configure()

.traversableResolver( new MyFastTraversableResolver() )

**.enableTraversableResolverResultCache( false )**

.buildValidatorFactory();

Validator validator = validatorFactory.getValidator();

**JavaDock**

Чтобы аннотации отображались в JavaDoc аннотированных элементов, сами типы аннотаций должны быть аннотированы мета-аннотацией @Documented. Это относится ко всем встроенным ограничениям и считается наилучшей практикой для любых пользовательских ограничений.

**Создание собственных аннотаций валидации**

***(только в случае многократного применения одной и той-же проверки)***

При создании собственной минимальной аннотации нужно будет:

1. определить 10 *(подчеркнутых)* параметров:

@Target({ FIELD, METHOD, PARAMETER, ANNOTATION\_TYPE, TYPE\_USE })

@Retention(RUNTIME)

@Constraint(validatedBy = CheckCaseValidator.class)

@Documented

@Repeatable(List.class)

public @interface CheckCase {

String message() default "{org.hibernate.validator.referenceguide.chapter06.CheckCase." +

"message}";

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

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

CaseMode value();

@Target({ FIELD, METHOD, PARAMETER, ANNOTATION\_TYPE })

@Retention(RUNTIME)

@Documented

@interface List {

CheckCase[] value();

}

2. Создать валидатор ограничения (и там тоже куча всего намучено)

3. Создать файл ValidationMessages.properties с сообщением об ошибке.

***Короче, геморрой конкретный*** + дополнительно усложним людям понимание и без того мутного java-вского кода.

Описание всего это смотреть в <https://docs.jboss.org/hibernate/stable/validator/reference/en-US/html_single/#validator-customconstraints-simple>

Далее приведены список аннотации из **базового** пакета (***есть еще доп.пакеты с аннотациями***):

**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-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 отрицательным, 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**

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.**

**@Valid**

Говорит о проверке также и другого собственного класса, объект которого мы собираемся создать у себя в классе. Ставиться перед необходимой аннотацией проверки.