HIBERNATE

# Hibernate Validator

Reference Guide

Version: 3.0.0.GA

# Table of Contents

# Preface

Annotations are a very convenient and elegant way to specify invariant constraints for a domain model. You can, for example, express that a property should never be null, that the account balance should be strictly positive, etc. These domain model constraints are declared in the bean itself by annotating its properties. A validator can then read them and check for constraint violations. The validation mechanism can be executed in different layers in your application without having to duplicate any of these rules (presentation layer, data access layer). Following the DRY principle, Hibernate Validator has been designed for that purpose.

Hibernate Validator works at two levels. First, it is able to check in-memory instances of a class for constraint violations. Second, it can apply the constraints to the Hibernate metamodel and incorporate them into the generated database schema.

Each constraint annotation is associated to a validator implementation responsible for checking the constraint on the entity instance. A validator can also (optionally) apply the constraint to the Hibernate metamodel, allowing Hibernate to generate DDL that expresses the constraint. With the appropriate event listener, you can execute the checking operation on inserts and updates done by Hibernate. Hibernate Validator is not limited to use with Hibernate. You can easily use it anywhere in your application as well as with any Java Persistence provider (entity listener provided).

When checking instances at runtime, Hibernate Validator returns information about constraint violations in an array of `InvalidValue` s. Among other information, the `InvalidValue` contains an error description message that can embed the parameter values bundle with the annotation (eg. length limit), and message strings that may be externalized to a `ResourceBundle` .

# Chapter 1. Defining constraints

## 1.1. What is a constraint?

A constraint is a rule that a given element (field, property or bean) has to comply to. The rule semantic is expressed by an annotation. A constraint usually has some attributes used to parameterize the constraints limits. The constraint applies to the annotated element.

## 1.2. Built in constraints

Hibernate Validator comes with some built-in constraints, which covers most basic data checks. As we'll see later, you're not limited to them, you can literally in a minute write your own constraints.

**Table 1.1. Built-in constraints**

| Annotation | Apply on | Runtime checking | Hibernate Metadata impact |
|---|---|---|---|
| @Length(min=, max=) | property (String) | check if the string length match the range | Column length will be set to max |
| @Max(value=) | property (numeric or string representation of a numeric) | check if the value is less than or equals to max | Add a check constraint on the column |
| @Min(value=) | property (numeric or string representation of a numeric) | check if the value is more than or equals to min | Add a check constraint on the column |
| @NotNull | property | check if the value is not null | Column(s) are not null |
| @NotEmpty | property | check if the string is not null nor empty. Check if the connection is not null nor empty | Column(s) are not null (for String) |
| @Past | property (date or calendar) | check if the date is in the past | Add a check constraint on the column |
| @Future | property (date or calendar) | check if the date is in the future | none |
| @Pattern(regex="regexp", flag=) or @Patterns( {@Pattern(...)} ) | property (string) | check if the property match the regular expression given a match flag (see `java.util.regex.Pattern` ) | none |
| @Range(min=, max=) | property (numeric or string representation of a numeric) | check if the value is between min and max (included) | Add a check constraint on the column |

| Annotation | Apply on | Runtime checking | Hibernate Metadata impact |
|---|---|---|---|
| @Size(min=, max=) | property (array, collection, map) | check if the element size is between min and max (included) | none |
| @AssertFalse | property | check that the method evaluates to false (useful for constraints expressed in code rather than annotations) | none |
| @AssertTrue | property | check that the method evaluates to true (useful for constraints expressed in code rather than annotations) | none |
| @Valid | property (object) | perform validation recursively on the associated object. If the object is a Collection or an array, the elements are validated recursively. If the object is a Map, the value elements are validated recursively. | none |
| @Email | property (String) | check whether the string is conform to the email address specification | none |
| @CreditCardNumber | property (String) | check whether the string is a well formated credit card number (derivative of the Luhn algorithm) | none |
| @Digits | property (numeric or string representation of a numeric) | check whether the property is a number having up to `integerDigits` integer digits and `fractionalDigits` fractonal digits | define column precision and scale |
| @EAN | property (string) | check whether the string is a properly formated EAN or UPC-A code | none |
| @Digits | property (numeric or string representation of a numeric) | check whether the property is a number having up to `integerDigits` integer digits and `fractionalDigits` fractonal digits | define column precision and scale |

## 1.3. Error messages

Hibernate Validator comes with a default set of error messages translated in about ten languages (if yours is not part of it, please sent us a patch). You can override those messages by creating a `ValidatorMessages.properties` or ( `ValidatorMessages_loc.properties` ) and override the needed keys. You can even add your own additional set of messages while writing your validator annotations. If Hibernate Validator cannot resolve a key from your resourceBundle nor from ValidatorMessage, it falls back to the default built-in values.

Alternatively you can provide a `ResourceBundle` while checking programmatically the validation rules on a bean or if you want a completly different interpolation mechanism, you can provide an implementation of `org.hibernate.validator.MessageInterpolator` (check the JavaDoc for more informations).

## 1.4. Writing your own constraints

Extending the set of built-in constraints is extremely easy. Any constraint consists of two pieces: the constraint *descriptor* (the annotation) and the constraint *validator* (the implementation class). Here is a simple user-defined descriptor:

```
@ValidatorClass(CapitalizedValidator.class)
@Target(METHOD)
@Retention(RUNTIME)
@Documented
public @interface Capitalized {
    CapitalizeType type() default Capitalize.FIRST;
    String message() default "has incorrect capitalization"
}
```

`type` is a parameter describing how the property should to be capitalized. This is a user parameter fully dependant on the annotation business.

`message` is the default string used to describe the constraint violation and is mandatory. You can hard code the string or you can externalize part/all of it through the Java ResourceBundle mechanism. Parameters values are going to be injected inside the message when the `{parameter}` string is found (in our example `Capitalization is not {type}` would generate `Capitalization is not FIRST` ), externalizing the whole string in `ValidatorMessages.properties` is considered good practice. See Error messages .

```
@ValidatorClass(CapitalizedValidator.class)
@Target(METHOD)
@Retention(RUNTIME)
@Documented
public @interface Capitalized {
    CapitalizeType type() default Capitalize.FIRST;
    String message() default "{validator.capitalized}";
}


#in ValidatorMessages.properties
validator.capitalized = Capitalization is not {type}
```

As you can see the {} notation is recursive.

To link a descriptor to its validator implementation, we use the `@ValidatorClass` meta-annotation. The validator class parameter must name a class which implements `Validator<ConstraintAnnotation>` .

We now have to implement the validator (ie. the rule checking implementation). A validation implementation can check the value of the a property (by implementing `PropertyConstraint` ) and/or can modify the hibernate mapping metadata to express the constraint at the database level (by implementing `PersistentClassConstraint` )

```
public class CapitalizedValidator
        implements Validator<Capitalized>, PropertyConstraint {
    private CapitalizeType type;

    //part of the Validator<Annotation> contract,
    //allows to get and use the annotation values
    public void initialize(Capitalized parameters) {
        type = parameters.type();
    }

    //part of the property constraint contract
    public boolean isValid(Object value) {
        if (value==null) return true;
        if ( !(value instanceof String) ) return false;
        String string = (String) value;
        if (type == CapitalizeType.ALL) {
            return string.equals( string.toUpperCase() );
        }
        else {
            String first = string.substring(0,1);
            return first.equals( first.toUpperCase();
        }
    }
}
```

The `isValid()` method should return false if the constraint has been violated. For more examples, refer to the built-in validator implementations.

We only have seen property level validation, but you can write a Bean level validation annotation. Instead of receiving the return instance of a property, the bean itself will be passed to the validator. To activate the validation checking, just annotated the bean itself instead. A small sample can be found in the unit test suite.

If your constraint can be applied multiple times (with different parameters) on the same property or type, you can use the following annotation form:

```
@Target(METHOD)
@Retention(RUNTIME)
@Documented
public @interface Patterns {
    Pattern[] value();
}

@Target(METHOD)
@Retention(RUNTIME)
@Documented
@ValidatorClass(PatternValidator.class)
public @interface Pattern {
    String regexp();
}
```

Basically an annotation containing the value attribute as an array of validator annotations.

## 1.5. Annotating your domain model

Since you are already familiar with annotations now, the syntax should be very familiar

```
public class Address {
    private String line1;
    private String line2;
    private String zip;
    private String state;
    private String country;
    private long id;

    // a not null string of 20 characters maximum
    @Length(max=20)
    @NotNull
    public String getCountry() {
        return country;
    }

    // a non null string
    @NotNull
    public String getLine1() {
        return line1;
    }

    //no constraint
    public String getLine2() {
        return line2;
    }

    // a not null string of 3 characters maximum
    @Length(max=3) @NotNull
    public String getState() {
        return state;
    }

    // a not null numeric string of 5 characters maximum
    // if the string is longer, the message will
    //be searched in the resource bundle at key 'long'
    @Length(max=5, message="{long}")
    @Pattern(regex="[0-9]+")
    @NotNull
    public String getZip() {
        return zip;
    }

    // should always be true
    @AssertTrue
    public boolean isValid() {
        return true;
    }

    // a numeric between 1 and 2000
    @Id @Min(1)
    @Range(max=2000)
    public long getId() {
        return id;
    }
}
```

While the example only shows public property validation, you can also annotate fields of any kind of visibility

```
@MyBeanConstraint(max=45
public class Dog {
    @AssertTrue private boolean isMale;
    @NotNull protected String getName() { ... };
    ...
}
```

You can also annotate interfaces. Hibernate Validator will check all superclasses and interfaces extended or implemented by a given bean to read the appropriate validator annotations.

```
public interface Named {
    @NotNull String getName();
    ...
}

public class Dog implements Named {

    @AssertTrue private boolean isMale;

    public String getName() { ... };

}
```

The name property will be checked for nullity when the Dog bean is validated.

# Chapter 2. Using the Validator framework

Hibernate Validator is intended to be used to implement multi-layered data validation, where constraints are expressed in a single place (the annotated domain model) and checked in various different layers of the application.

This chapter will cover Hibernate Validator usage for different layers

## 2.1. Database schema-level validation

Out of the box, Hibernate Annotations will translate the constraints you have defined for your entities into mapping metadata. For example, if a property of your entity is annotated `@NotNull`, its columns will be declared as `not null` in the DDL schema generated by Hibernate.

Using hbm2ddl, domain model constraints will be expressed into the database schema.

If, for some reason, the feature needs to be disabled, set `hibernate.validator.apply_to_ddl` to `false`.

## 2.2. ORM integration

Hibernate Validator integrates with both Hibernate and all pure Java Persistence providers

### 2.2.1. Hibernate event-based validation

Hibernate Validator has two built-in Hibernate event listeners. Whenever a `PreInsertEvent` or `PreUpdateEvent` occurs, the listeners will verify all constraints of the entity instance and throw an exception if any constraint is violated. Basically, objects will be checked before any inserts and before any updates made by Hibernate. This includes changes applied by cascade! This is the most convenient and the easiest way to activate the validation process. On constraint violation, the event will raise a runtime `InvalidStateException` which contains an array of `InvalidValues` describing each failure.

If Hibernate Validator is present in the classpath, Hibernate Annotations (or Hibernate EntityManager) will use it transparently. If, for some reason, you want to disable this integration, set `hibernate.validator.autoregister_listeners` to false

#### Note

If the beans are not annotated with validation annotations, there is no runtime performance cost.

In case you need to manually set the event listeners for Hibernate Core, use the following configuration in `hibernate.cfg.xml`:

```
<hibernate-configuration>
    ...
    <event type="pre-update">
        <listener
          class="org.hibernate.validator.event.ValidateEventListener"/>
    </event>
    <event type="pre-insert">
        <listener
          class="org.hibernate.validator.event.ValidateEventListener"/>
    </event>
```

```
</hibernate-configuration>
```

### 2.2.2. Java Persistence event-based validation

Hibernate Validator is not tied to Hibernate for event based validation: a Java Persistence entity listener is available. Whenever an listened entity is persisted or updated, Hibernate Validator will verify all constraints of the entity instance and throw an exception if any constraint is violated. Basically, objects will be checked before any inserts and before any updates made by the Java Persistence provider. This includes changes applied by cascade! On constraint violation, the event will raise a runtime `InvalidStateException` which contains an array of `InvalidValues` describing each failure.

Here is how to make a class validatable:

```
@Entity
@EntityListeners( JPAValidateListener.class )
public class Submarine {
    ...
}
```

**Note**

Compared to the Hibernate event, the Java Persistence listener has two drawbacks. You need to define the entity listener on every validatable entity. The DDL generated by your provider will not reflect the constraints.

## 2.3. Application-level validation

Hibernate Validator can be applied anywhere in your application code.

```
ClassValidator personValidator = new ClassValidator( Person.class );
ClassValidator addressValidator = new ClassValidator( Address.class, ResourceBundle.getBundle("message

InvalidValue[] validationMessages = addressValidator.getInvalidValues(address);
```

The first two lines prepare the Hibernate Validator for class checking. The first one relies upon the error messages embedded in Hibernate Validator (see Error messages), the second one uses a resource bundle for these messages. It is considered a good practice to execute these lines once and cache the validator instances.

The third line actually validates the `Address` instance and returns an array of `InvalidValues`. Your application logic will then be able to react to the failure.

You can also check a particular property instead of the whole bean. This might be useful for property per property user interaction

```
ClassValidator addressValidator = new ClassValidator( Address.class, ResourceBundle.getBundle("message

//only get city property invalid values
InvalidValue[] validationMessages = addressValidator.getInvalidValues(address, "city");

//only get potential city property invalid values
InvalidValue[] validationMessages = addressValidator.getPotentialInvalidValues("city", "Paris")
```

## 2.4. Presentation layer validation

When working with JSF and JBoss Seam™, one can triggers the validation process at the presentation layer using Seam's JSF tags `<s:validate>` and `<s:validateAll/>`, letting the constraints be expressed on the model, and the violations presented in the view

```
<h:form>
    <div>
        <h:messages/>
    </div>
    <s:validateAll>
        <div>
            Country:
            <h:inputText value="#{location.country}" required="true"/>
        </div>
        <div>
            Zip code:
            <h:inputText value="#{location.zip}" required="true"/>
        </div>
        <div>
            <h:commandButton/>
        </div>
    </s:validateAll>
</h:form>
```

Going even further, and adding Ajax4JSF™ to the loop will bring client side validation with just a couple of additional JSF tags, again without validation definition duplication.

Check the JBoss Seam [http://www.jboss.com/products/seam] documentation for more information.

## 2.5. Validation informations

As a validation information carrier, hibernate provide an array of `InvalidValue`. Each `InvalidValue` has a buch of methods describing the individual issues.

`getBeanClass()` retrieves the failing bean type

`getBean()`retrieves the failing instance (if any ie not when using `getPotentianInvalidValues()`)

`getValue()` retrieves the failing value

`getMessage()` retrieves the proper internationalized error message

`getRootBean()` retrieves the root bean instance generating the issue (useful in conjunction with `@Valid`), is null if getPotentianInvalidValues() is used.

`getPropertyPath()` retrieves the dotted path of the failing property starting from the root bean