



# **JBoss Enterprise Web Platform 5 Hibernate Validator Reference Guide**

---

for Use with JBoss Enterprise Web Platform 5  
Edition 5.1.1

Red Hat Documentation Group

# JBoss Enterprise Web Platform 5 Hibernate Validator Reference Guide

---

for Use with JBoss Enterprise Web Platform 5  
Edition 5.1.1

Red Hat Documentation Group

## Legal Notice

Copyright © 2011 Red Hat, Inc.

This document is licensed by Red Hat under the [Creative Commons Attribution-ShareAlike 3.0 Unported License](#). If you distribute this document, or a modified version of it, you must provide attribution to Red Hat, Inc. and provide a link to the original. If the document is modified, all Red Hat trademarks must be removed.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, MetaMatrix, Fedora, the Infinity Logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux® is the registered trademark of Linus Torvalds in the United States and other countries.

Java® is a registered trademark of Oracle and/or its affiliates.

XFS® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack® Word Mark and OpenStack Logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

## Abstract

The Hibernate Validator Reference Guide for JBoss Enterprise Web Platform 5 and its patch releases.

# Table of Contents

<b>Preface</b> .....	<b>3</b>
1. Document Conventions	3
1.1. Typographic Conventions	3
1.2. Pull-quote Conventions	4
1.3. Notes and Warnings	5
2. Getting Help and Giving Feedback	5
2.1. Do You Need Help?	5
2.2. Give us Feedback	6
<b>Chapter 1. Introduction</b> .....	<b>7</b>
<b>Chapter 2. Defining constraints</b> .....	<b>8</b>
2.1. What is a constraint?	8
2.2. Built in constraints	8
2.3. Error messages	10
2.4. Writing your own constraints	10
2.5. Annotating your domain model	12
<b>Chapter 3. Using the Validator framework</b> .....	<b>15</b>
3.1. Database schema-level validation	15
3.2. ORM integration	15
3.2.1. Hibernate event-based validation	15
3.2.2. Java Persistence event-based validation	16
3.3. Application-level validation	16
3.4. Presentation layer validation	17
3.5. Validation informations	17
<b>Revision History</b> .....	<b>19</b>



# Preface

## 1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

In PDF and paper editions, this manual uses typefaces drawn from the [Liberation Fonts](#) set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later include the Liberation Fonts set by default.

### 1.1. Typographic Conventions

Four typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

#### **Mono-spaced Bold**

Used to highlight system input, including shell commands, file names and paths. Also used to highlight keys and key combinations. For example:

To see the contents of the file **my\_next\_bestselling\_novel** in your current working directory, enter the **cat my\_next\_bestselling\_novel** command at the shell prompt and press **Enter** to execute the command.

The above includes a file name, a shell command and a key, all presented in mono-spaced bold and all distinguishable thanks to context.

Key combinations can be distinguished from an individual key by the plus sign that connects each part of a key combination. For example:

Press **Enter** to execute the command.

Press **Ctrl+Alt+F2** to switch to a virtual terminal.

The first example highlights a particular key to press. The second example highlights a key combination: a set of three keys pressed simultaneously.

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph will be presented as above, in **mono-spaced bold**. For example:

File-related classes include **filesystem** for file systems, **file** for files, and **dir** for directories. Each class has its own associated set of permissions.

#### **Proportional Bold**

This denotes words or phrases encountered on a system, including application names; dialog box text; labeled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

Choose **System** → **Preferences** → **Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, select the **Left-handed mouse** check box and click **Close** to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).

To insert a special character into a **gedit** file, choose **Applications** → **Accessories** →

**Character Map** from the main menu bar. Next, choose **Search** → **Find...** from the **Character Map** menu bar, type the name of the character in the **Search** field and click **Next**. The character you sought will be highlighted in the **Character Table**. Double-click this highlighted character to place it in the **Text to copy** field and then click the **Copy** button. Now switch back to your document and choose **Edit** → **Paste** from the **gedit** menu bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all presented in proportional bold and all distinguishable by context.

### ***Mono-spaced Bold Italic*** or ***Proportional Bold Italic***

Whether mono-spaced bold or proportional bold, the addition of italics indicates replaceable or variable text. Italics denotes text you do not input literally or displayed text that changes depending on circumstance. For example:

To connect to a remote machine using ssh, type **ssh *username@domain.name*** at a shell prompt. If the remote machine is **example.com** and your username on that machine is john, type **ssh *john@example.com***.

The **mount -o remount *file-system*** command remounts the named file system. For example, to remount the **/home** file system, the command is **mount -o remount */home***.

To see the version of a currently installed package, use the **rpm -q *package*** command. It will return a result as follows: ***package-version-release***.

Note the words in bold italics above — username, domain.name, file-system, package, version and release. Each word is a placeholder, either for text you enter when issuing a command or for text displayed by the system.

Aside from standard usage for presenting the title of a work, italics denotes the first use of a new and important term. For example:

Publican is a *DocBook* publishing system.

## 1.2. Pull-quote Conventions

Terminal output and source code listings are set off visually from the surrounding text.

Output sent to a terminal is set in **mono-spaced roman** and presented thus:

```
books      Desktop  documentation  drafts  mss    photos  stuff  svn
books_tests Desktop1  downloads      images  notes  scripts svgs
```

Source-code listings are also set in **mono-spaced roman** but add syntax highlighting as follows:

```
package org.jboss.book.jca.ex1;

import javax.naming.InitialContext;

public class ExClient
{
    public static void main(String args[])
        throws Exception
    {
        InitialContext iniCtx = new InitialContext();
        Object          ref    = iniCtx.lookup("EchoBean");
        EchoHome        home   = (EchoHome) ref;
        Echo            echo    = home.create();

        System.out.println("Created Echo");

        System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
    }
}
```

### 1.3. Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.



#### Note

Notes are tips, shortcuts or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.



#### Important

Important boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring a box labeled 'Important' will not cause data loss but may cause irritation and frustration.



#### Warning

Warnings should not be ignored. Ignoring warnings will most likely cause data loss.

## 2. Getting Help and Giving Feedback

### 2.1. Do You Need Help?

If you experience difficulty with a procedure described in this documentation, visit the Red Hat Customer Portal at <http://access.redhat.com>. Through the customer portal, you can:

- search or browse through a knowledgebase of technical support articles about Red Hat products.
- submit a support case to Red Hat Global Support Services (GSS).
- access other product documentation.

Red Hat also hosts a large number of electronic mailing lists for discussion of Red Hat software and technology. You can find a list of publicly available mailing lists at <https://www.redhat.com/mailman/listinfo>. Click on the name of any mailing list to subscribe to that list or to access the list archives.

## 2.2. Give us Feedback

If you find a typographical error, or know how this guide can be improved, we would love to hear from you. Submit a report in Bugzilla against the product **JBoss Enterprise Application Platform 5** and the component **doc-Hibernate\_Validator\_Ref\_Guide**. The following link will take you to a pre-filled bug report for this product: <http://bugzilla.redhat.com/>.

Fill out the following template in Bugzilla's **Description** field. Be as specific as possible when describing the issue; this will help ensure that we can fix it quickly.

Document URL:

Section Number and Name:

Describe the issue:

Suggestions for improvement:

Additional information:

Be sure to give us your name so that you can receive full credit for reporting the issue.

## Chapter 1. Introduction

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 2. Defining constraints

### 2.1. 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.

### 2.2. 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 2.1. 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 <b>java.util.regex.Pattern</b> )	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
@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	none

		an array, the elements are validated recursively. If the object is a Map, the value elements are validated recursively.	
@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 formatted credit card number (derivative of the Luhn algorithm)	none
@Digits(integerDigits=1)	property (numeric or string representation of a numeric)	check whether the property is a number having up to <b>integerDigits</b> integer digits and <b>fractionalDigits</b> fractional digits	define column precision and scale
@EAN	property (string)	check whether the string is a properly formatted EAN or UPC-A code	none

## 2.3. 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 send 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 completely different interpolation mechanism, you can provide an implementation of **org.hibernate.validator.MessageInterpolator** (check the JavaDoc for more informations).

## 2.4. 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 [Section 2.3, “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.

## 2.5. 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 3. 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

### 3.1. 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**.

### 3.2. ORM integration

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

#### 3.2.1. 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>
```

### 3.2.2. 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.

## 3.3. 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("messages", Locale.ENGLISH) );

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 [Section 2.3, "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("messages", Locale.ENGLISH) );

//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");
```

### 3.4. 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](#) documentation for more information.

### 3.5. Validation informations

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

**getBeanClass()** retrieves the failing bean type

**getBean()** retrieves the failing instance (if any ie not when using **getPotentialInvalidValues()**)

**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 **getPotentialInvalidValues()** is used.

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

## Revision History

<b>Revision 5.1.1-104.400</b>	<b>2013-10-31</b>	<b>Rüdiger Landmann</b>
Rebuild with publican 4.0.0		
<b>Revision 5.1.1-104</b>	<b>2012-07-18</b>	<b>Anthony Towns</b>
Rebuild for Publican 3.0		
<b>Revision 5.1.1-100</b>	<b>Mon Jul 18 2011</b>	<b>Jared Morgan</b>
Incorporated changes for JBoss Enterprise Web Platform 5.1.1 GA. For information about documentation changes to this guide, refer to <i>Release Notes 5.1.1</i> .		