

# JaValid Documentation

Version: 1.1  
Author: M.Reuvers  
Date: September, 2008

# Contents

1	Introduction.....	1
2	Installation.....	2
2.1	Dependencies.....	2
3	Configuration.....	3
3.1	Core configuration.....	3
3.2	Spring Configuration.....	6
3.3	JSF Configuration.....	6
3.4	Database Configuration.....	6
4	Core annotations.....	8
4.1	Metadata Annotations.....	8
4.1.1	@ValidateDefinition.....	8
4.1.2	@Lookup.....	8
4.1.3	@BeanLookup.....	9
4.1.4	@SpringLookup.....	10
4.1.5	@JvGroup.....	10
4.1.6	@JvMethod.....	11
4.1.7	@JiParam.....	11
4.2	Validation annotations.....	13
4.2.1	Plural annotations.....	13
4.2.2	@NotNull.....	13
4.2.3	@NotEmpty.....	14
4.2.4	@BetweenLength.....	14
4.2.5	@MaxLength.....	14
4.2.6	@MinLength.....	15
4.2.7	@MinValue.....	15
4.2.8	@MaxValue.....	15
4.2.9	@LovConstraint.....	16
4.2.10	@RegularExpression.....	18
4.2.11	@ValidateList.....	18
4.2.12	@ValidateMap.....	18
4.2.13	@CollectionSize.....	18
4.2.14	@DateCheck.....	19
4.3	Add custom validation annotations.....	20
4.3.1	Create the annotation.....	20
4.3.2	Create the validator implementing JvalidValidator.....	21
4.3.3	Add your annotation to the configuration.....	21
4.3.4	Plural annotations.....	22
4.4	Annotation value resolver.....	23
4.4.1	Introduction.....	23
4.4.2	Configuration.....	24
4.4.3	Key usage.....	25
5	Validator usage.....	26
5.1	Interface.....	26
5.2	Implementation.....	27
5.3	Callback handler.....	28
5.4	Exception handling.....	29
5.5	Validation Messages (i18n).....	29
5.6	Singleton, threading.....	29
6	Spring extension.....	31
6.1	Configuration and usage.....	31
6.2	Spring Validator.....	32
6.3	Message converter.....	34
7	JSF extension.....	35
7.1	Introduction.....	35
7.2	Integrating validation in the Update Model Values phase.....	35
7.2.1	Web application configuration.....	36
7.2.1.1	Let the JvJsfContextListener load a validator directly.....	36
7.2.1.2	Locate validator as a Spring/JSF bean.....	37
7.2.2	JSF Configuration file.....	37
7.2.2.1	Resource-bundle tag.....	38
7.2.2.2	View tag.....	38

7.2.2.3 Validation-rule tag.....	38
7.3 Direct JSF Component validation.....	39
7.3.1 validateAll tag.....	40
7.3.2 validate tag.....	40
7.4 Some handy methods.....	41
7.4.1 org.javalid.external.jsf.JsfSupport.....	41
7.4.2 org.javalid.external.jsf.JsfMessageConverter.....	41
8 Database extension.....	42
8.1 Configuration and usage.....	42
8.1.1 Datasource tag.....	43
8.1.2 Connection tag.....	44
8.1.3 Sql-query tag.....	44
8.2 Validation annotations.....	45
8.2.1 @DbNumCheck.....	45
8.3 Expressions .....	46
9 Add custom extension.....	48
10 Examples.....	50
11 Release notes.....	51
12 Upgrading JaValid.....	53
12.1 Upgrade 1.0 to 1.1.....	53
13 Thanks.....	54



# 1 Introduction

JaValid (Java Validation) as the name suggests provides a way for validating Java objects. The framework aims at providing a robust solution by not only providing standard validation, but also allows custom validation by integrating with Spring, JSF and Facelets. It also offers an extension to integrate validation with a database since 1.1.

The idea for this framework originates from a project using techniques like Spring, JSF and Hibernate (ejb3). Several validation frameworks/options were considered such as Hibernate Validator, but they lacked the flexibility needed for this specific project. So it was decided to build this new framework. The project has been refactored and a lot of extra documentation was added prior to its first release.

You can use the core of the framework as stand-alone, which provides all the validation that is needed. However extensions can be used to make your life easier or to use a special way of validation and/or use certain message converters.

Hint: All classes of the framework are well-documented, by javadoc – so if you need to know something about an annotation, interface, method etc. and cannot find it in this document, check it out! For example usages, check out the website [www.javalid.org](http://www.javalid.org) as well as the unit tests that ship in the distribution.

Have fun using JaValid!

## 2 Installation

Download the latest release of the framework on <http://www.javalid.org> and extract the file.

The binary distribution currently ships with 4 .jar files (where xx is the version number of the distribution)

- javalid-core-xx.jar
- javalid-spring-dep-xx.jar
- javalid-jsf-dep-xx.jar
- javalid-db-dep-xx.jar

The core file contains all classes needed for validation and can be used standalone; the other .jar files are optional depending on your needs.

If you need special validation for Spring (this may be the case if you use the Spring Framework in your project) you can use javalid-spring-dep.jar as an extension, which provides hooks for integration with Spring.

If you need integrated Java Server Faces (JSF) or Facelets validation; or simply support for quickly converting JaValid validation messages to JSF messages and stuff like that, you need javalid-jsf-dep.jar.

The extension javalid-db-dep.jar allows you to integrate validation with your database (through the use of your existing datasource(s) or allows you to configure a datasource yourself).

Whatever you choose, the .jar files should ship with your application, for example in your /WEB-INF/lib directory. As a side note: You can use the framework in any application, not just a web-application.

### 2.1 Dependencies

The core jar file (javalid-core.jar) depends on

- *Log4j* (<http://log4j.apache.org>)
- *Juel* (<http://juel.sourceforge.net>)

You must add a recent version of these to your application's classpath. Since 1.1 both these libraries ship within the binary release as well, so you can use these if you want.

**Important:** Be careful if your application server already has an EL implementation (e.g. Tomcat 6 has), in that case you must deploy the juel-xx-impl.jar file or you may get classloading issues (the impl contains only juel files, the juel-xx.jar includes both juel and javax.el.\* classes). In any case you need only one of the juel files!

The spring extension (javalid-spring-dep.jar) depends on the core Spring libraries, but you will have these automatically once you use Spring in your project.

The JSF (Java Server Faces) extension depends on a JSF implementation (e.g. Sun's Reference Implementation or Apache Myfaces).

Finally, the database extension depends on Apache Jakarta Commons libraries:

- commons-dbcp-1.2.2.jar
- commons-logging-1.1.jar
- commons-pool-1.3.jar

You can download them from <http://commons.apache.org> directly, but for your convenience they ship within the binary distribution as well.

Of course it also depends on your database driver, so you are responsible to provide that.

## 3 Configuration

This chapter describes the (optional) configuration for the framework. The core configuration is discussed first, followed by how to enable Spring integration. The next part discusses the configuration when you wish to use JSF validation and the last part discusses the database integration.

### 3.1 Core configuration

The core can be configured using an xml file, by default the framework ships with its own configuration file, which is used if you don't change anything. In most cases this is enough if you just use the core part of the framework and need no extensions enabled. This file can be found in the package: `org/jvalid/core/config/` with the name `jav- config.xml`. However if you need to make changes you can provide your own configuration file.

Note that since 1.1 this file has changed a lot, the 1.0 file will still work – but only for the core. However, you are highly advised to upgrade to this new version! The default xml file looks like this:

```

<?xml version="1.0" encoding="utf-8" ?>
<jv annotations-on-element="method">

  <!-- registered annotations and their respective implementations -->
  <annotations>
    <annotation annotation-class="org.javalid.annotations.validation.NotNull"
      validator-class="org.javalid.core.validator.NotNullValidatorImpl"
      supports-plural="false" />
    <annotation annotation-class="org.javalid.annotations.validation.NotEmpty"
      validator-class="org.javalid.core.validator.NotEmptyValidatorImpl"
      supports-plural="false" />
    <annotation annotation-class="org.javalid.annotations.validation.MinLength"
      validator-class="org.javalid.core.validator.MinLengthValidatorImpl"
      supports-plural="true" />
    <annotation annotation-class="org.javalid.annotations.validation.MaxLength"
      validator-class="org.javalid.core.validator.MaxLengthValidatorImpl"
      supports-plural="true" />
    <annotation annotation-class="org.javalid.annotations.validation.BetweenLength"
      validator-class="org.javalid.core.validator.BetweenLengthValidatorImpl"
      supports-plural="true" />
    <annotation annotation-class="org.javalid.annotations.validation.MinValue"
      validator-class="org.javalid.core.validator.MinValueValidatorImpl"
      supports-plural="true" />
    <annotation annotation-class="org.javalid.annotations.validation.MaxValue"
      validator-class="org.javalid.core.validator.MaxValueValidatorImpl"
      supports-plural="true" />
    <annotation annotation-class="org.javalid.annotations.validation.LovConstraint"
      validator-class="org.javalid.core.validator.LovConstraintValidatorImpl"
      supports-plural="true" />
    <annotation annotation-class="org.javalid.annotations.validation.ValidateMap"
      validator-class="org.javalid.core.validator.ValidateMapValidatorImpl"
      supports-plural="true" />
    <annotation annotation-class="org.javalid.annotations.validation.ValidateList"
      validator-class="org.javalid.core.validator.ValidateListValidatorImpl"
      supports-plural="true" />
    <annotation annotation-class="org.javalid.annotations.validation.RegularExpression"
      validator-class="org.javalid.core.validator.RegularExpressionValidatorImpl"
      supports-plural="true" />
    <annotation annotation-class="org.javalid.annotations.validation.CollectionSize"
      validator-class="org.javalid.core.validator.CollectionSizeValidatorImpl"
      supports-plural="true" />

    <annotation annotation-class="org.javalid.annotations.validation.DateCheck"
      validator-class="org.javalid.core.validator.DateCheckValidatorImpl"
      supports-plural="true" />

    <!--
      Database extension annotations, enable if you turn database support on. Since 1.1.
    -->
    <annotation annotation-class="org.javalid.external.db.annotations.validation.DbNumCheck"
      validator-class="org.javalid.external.db.validator.DbNumCheckValidatorImpl"
      supports-plural="true" />

    -->
  </annotations>

  <!--
  <extensions>
    <extension name="spring" extension-loader-class="org.javalid.external.spring.extension.JavalidExtensionSpringImpl">
      <parameters>
        <parameter name="validatorClass" value="org.javalid.external.spring.validator.SpringValidatorExtImpl" />
        <parameter name="lookupClass" value="org.javalid.external.spring.lookup.SpringLookupExtImpl" />
      </parameters>
    </extension>
    <extension name="jsf" extension-loader-class="org.javalid.external.jsf.extension.JavalidExtensionJsfImpl">
      <parameters>
        <parameter name="validatorClass" value="org.javalid.external.jsf.validator.JsfValidatorExtImpl" />
        <parameter name="lookupClass" value="org.javalid.external.jsf.lookup.JsfLookupExtImpl" />
        <parameter name="validatorExpr" value="#{validatorBean}" />
      </parameters>
    </extension>
    <extension name="database" extension-loader-class="org.javalid.external.db.extension.JavalidExtensionDatabaseImpl">
      <parameters>
        <parameter name="configFile" value="org/javalid/external/db/config/javalid-db-example.xml" />
      </parameters>
    </extension>
  </extensions>

  -->

  <!-- <check-for-proxy in-class-name="EnhancerByCGLIB" /> -->
</jv>

```

If you wish to make your own configuration file, copy this file to your own xml file as a basis (use the one that ships with the distribution, not this example as it might be outdated!). You can then tell the validator to use your configuration file and not the default one.

#### **<jv annotations-on-element="method">**

This is a critical choice; this tells the framework that the validation annotations can be either found on method or fields. Valid values are: method or field

Make sure to choose this correctly or validation will fail!

The following tags are important to understand:

#### **<annotations></annotations>**

Contains all *validation* annotations with their respective validation implementation. Note that the NotNull and NotEmpty annotations are *required*. It is likely these two will be moved out of the configuration file in a later release to prevent confusion.

If you have your own validation annotation and wish to make it known by the framework, just add a new definition here like:

```
<annotation annotation-class="mypackage.MyAnnotation" validator-  
class="mypackage.MyAnnotationValidatorImpl" supports-plural="true | false" annotation-class-  
plural="mypackage.myPluralAnnotation" />
```

This tells the framework that a new annotation is available as mypackage.MyAnnotation (full classname) and its validator is known as mypackage.MyAnnotationValidatorImpl.

The new parameters 'supports-plural' and 'annotation-class-plural' are optional. If supports-plural is set to true you must have an annotation which supports multiple definitions, like for instance the new plural annotation @MinLengths(..), which supports multiple @MinLength annotations. You can leave out annotation-class-plural if your plural class is called the same as the annotation defined here with an extra s (@MinLength → @MinLengths). If this is not the case you must specify your plural annotation. The plural annotation must have a property like: MinLength[] values(); So an array of the annotation you support. For more information about this see chapter 4.2.1.

**Important:** Assure your validator class can be used by multiple threads as the framework loads it as a 'singleton' (thus only one instance is created). The easiest way to do this is to make sure you don't use changing data (members) in your validator.

#### **<extensions></extensions>**

The extensions tag is new since 1.1 and allows you to enable/disable extensions. The original spring / jsf extensions have been moved to this new architecture as well. Beside enabling existing extensions, it allows you to add your own new extensions in a very simple way. To learn more about this see chapter 9.

#### **<check-for-proxy in-class-name="EnhancerByCGLIB" />**

This is an optional tag (thus can be left out). However if you use frameworks like for instance, Hibernate, Spring etc. which may proxy their objects, you might just need this tag. Setting this tag assures that lookups of methods succeed in the case of proxied objects. The parameter *in-class-name* must be set to a part of a proxied classname, for instance the CGLIB library creates a proxy class with a name that contains 'EnhancerByCGLIB'.

If the validator encounters a proxied class, it instead uses its parent (or that parent etc) until it is a normal class again. But it will only do this if this tag is set. You can specify multiple names, by separating them with a comma.

**Warning:** Not setting it, while using proxied objects may result in validation not being fired or other weird behaviour might happen (as annotations do not inherit, which is the original cause for this problem).

That is all for the core configuration, not that hard was it – just remember that it is here that you add new annotation validators and can enable/disable extensions.

## 3.2 Spring Configuration

Since 1.1 you can enable the Spring configuration by enabling its extension like:

```
<extension name="spring"
  extension-loader-
class="org.javalid.external.spring.extension.JavalidExtensionSpringImpl">
  <parameters>
    <parameter name="validatorClass"
      value="org.javalid.external.spring.validator.SpringValidatorExtImpl"/>
    <parameter name="lookupClass"
      value="org.javalid.external.spring.lookup.SpringLookupExtImpl"/>
  </parameters>
</extension>
```

Extensions can be passed parameters, in this case the validatorClass and lookupClass parameter are set. It tells the framework to use given validator and lookup class. These implementations ship in the javalid-spring-dep.jar.

If for some reason you wish to provide different implementations, feel free to do so, your classes must implement: org.javalid.core.extlookup.SpringLookupExt and org.javalid.core.extvalidator.SpringValidatorExt respectively. Note that they both must have an empty public constructor.

Note: The extension name must be spring and may not change!

## 3.3 JSF Configuration

Since 1.1 you can enable the JSF configuration by enabling its extension like:

```
<extension name="jsf"
  extension-loader-
class="org.javalid.external.jsf.extension.JavalidExtensionJsfImpl">
  <parameters>
    <parameter name="validatorClass"
      value="org.javalid.external.jsf.validator.JsfValidatorExtImpl"/>
    <parameter name="lookupClass"
      value="org.javalid.external.jsf.lookup.JsfLookupExtImpl"/>
    <parameter name="validatorExpr" value="#{validatorBean}"/>
  </parameters>
</extension>
```

Extensions can be passed parameters, in this case the validatorClass and lookupClass parameter are set. It tells the framework to use given validator and lookup class. These implementations ship in the javalid-jsf-dep.jar.

Just as with Spring you can also provide custom implementations if needed. Your classes respectively need to implement org.javalid.external.jsf.validator.JsfValidatorExt and org.javalid.external.jsf.lookup.JsfLookupExt. They both require an empty public constructor.

Note: The extension name must be jsf and may not change!

## 3.4 Database Configuration

JaValid 1.1 ships with a new powerful extension, the database extension. This paragraph only tells you how to enable it, to learn more about it see chapter 8.

```
<extension name="database"
  extension-loader-
  class="org.javalid.external.db.extension.JavalidExtensionDatabaseImpl">
  <parameters>
    <parameter name="configFile"
      value="org/javalid/external/db/config/javalid-db-example.xml"/>
  </parameters>
</extension>
```

The database extension expects only one parameter, the location of a separate config file. Note this is only a sample file, you must really provide one yourself.

The database extension allows you to perform checks against the database of your choice. To learn more on how to use it, see chapter 8.

## 4 Core annotations

This chapter describes what the core of the framework is all about: Validation. We will discuss how to use the framework using its core annotations. First the metadata annotations are discussed followed by an explanation of the default available validation annotations.

### 4.1 Metadata Annotations

One could say the framework nearly 'lives' on its annotations. You can use (metadata) annotations to tell the framework what model objects it must consider for validation and on how to validate an object. Validation can be applied to nearly any java object, but usually you will annotate your data model objects. These can be ordinary POJO's (Plain Old Java Objects) or EJB's for instance.

A few 'rules' for the following discussion, underlined text represents a property of the annotation we discuss, *italic* text represents the default value of a property (if you don't set it), a | (pipeline) represents the logic 'or'-operator (thus one of the pipelined options).

The framework considers an object for validation only if it is annotated by the following annotation:

#### 4.1.1 @ValidateDefinition

Specify this property at class level.

```
@ValidateDefinition
public class Example {
    ...
}
```

primaryGroup=*"1"* | String

This property denotes the default group name for validation that is used by the validator. The framework supports validation in groups, hence it is possible to define multiple groups and apply different validations if needed depending on your needs. This default group name is used when you don't specify any group name when using the AnnotationValidator.

This annotation has a few more of optional properties which are discussed below:

validationType=*TYPE\_NORMAL* | TYPE\_LOOKUP

This one represents how the annotated object is validated, by default this is normal validation (you usually pick this one), but you can also validate this object in a custom way by TYPE\_LOOKUP. If this is chosen you must specify the lookup property of this annotation. Lookup defines that you want to use something else, such as a Javabean or Spring bean.

forceDoubleValidation=true | *false*. Is relevant only if you change validationType to TYPE\_LOOKUP. If set to true, you can apply both the special validation (through lookup) and normal validation of a group.

lookup=@Lookup(..)

The type of lookup you wish to use is defined through @Lookup, which is discussed below.

#### 4.1.2 @Lookup

This annotation is used to define what type of Lookup is used. It can be used in different annotations, for instance @ValidateDefinition and @LovConstraint use it.

type=*LOOKUP\_NORMAL\_BEAN* | LOOKUP\_SPRING\_BEAN

This property is required. Specify LOOKUP\_NORMAL\_BEAN in the case you wish to use a normal Javabean to perform validation (or anything else, depending on the context where this annotation is used). In this case you must also define property beanLookup. Specify LOOKUP\_SPRING\_BEAN in the case you wish to use a Spring bean, you must specify property springLookup then.

beanLookup=@BeanLookup(..)

Required, if type is set to LOOKUP\_NORMAL\_BEAN. More can be found below.

springLookup=@SpringLookup(..)

Required, if type is set to LOOKUP\_SPRING\_BEAN. More can be found below.

### 4.1.3 @BeanLookup

This annotation is used as a property inside the @Lookup annotation. You can specify a java object (bean) with it, which will perform the actual validation. This is useful where you have such complex validation needs and cannot do them in the normal way, but want them to be applied automatically by the framework by an ordinary java object. For example:

```
@ValidateDefinition (
    primaryGroup="1",
    validationType=ValidateDefinition.TYPE_LOOKUP,
    lookup=@Lookup (
        type=Lookup.LOOKUP_NORMAL_BEAN,
        beanLookup=@BeanLookup (
            beanClass=BeanLookupValidation.class,
            method=@JvMethod (
                name="validate",
                params={
                    @JvParam (
                        valueRetrievalMode=JvParam.MODE_CURRENT_OBJECT
                    ),
                    @JvParam (
                        valueRetrievalMode=JvParam.MODE_CURRENT_PATH
                    )
                }
            )
        )
    )
)
```

This example declares that we wish to use an ordinary java class to perform the actual validation for the object annotated. The beanClass points to the actual java class and which method to perform the validation with. As you can see you can fully specify the method yourself, in this case it is a validation method. But it could be something else when for instance used in @LovConstraint.

The @BeanLookup annotation has the following properties:

beanClass=Full className. The class of the bean that will perform the actual validation. Required.

method=The method to use for validation, this is an annotation itself, more about it later. Required.

cacheBean=true | false. If set to true the bean is cached (it is initialized once and then cached). Set to false if you do not want the bean to be cached.

Note: Your bean must be able to handle multiple threads (most objects can as long as you do not store data in the class itself) if you cache the bean.

## 4.1.4 @SpringLookup

This annotation can be used as a property inside the @Lookup annotation. You can specify a spring bean with it, which will perform the actual validation. Most often you will use this for already existing validation logic or complex needs. An example:

```
@ValidateDefinition (
    primaryGroup="1",
    validationType=ValidateDefinition.TYPE_LOOKUP,
    forceDoubleValidation=false,
    lookup=@Lookup (
        type=Lookup.LOOKUP_SPRING_BEAN,
        springLookup=@SpringLookup(
            name="SimpleServiceBean",
            method=@JvMethod(
                name="validateEmployee",
                params={
                    @JvParam(valueRetrievalMode=JvParam.MODE_CURRENT_OBJECT),
                    @JvParam(valueRetrievalMode=JvParam.MODE_CURRENT_PATH)
                }
            )
        )
    )
)
```

This example specifies that for this annotated object a Spring bean must be used for its validation. The name represents the name of the Spring bean, and the method represents the method that must be called and with what parameters. Again depending on the context where @SpringLookup is used, you could define a different type of method depending on the requirements there.

@SpringLookup has the following properties:

name=Spring bean id. The name of the Spring bean, this should be the name (id) of an existing bean declared in your spring configuration. Required.

method=The method to use for validation, this is an annotation itself, more about it later. Required.

## 4.1.5 @JvGroup

This annotation is used on method level and must be specified on 'get' methods. Using this annotation you can tell the framework that this method should be considered for validation. Whether validation takes place at all depends on this annotation. Forgetting it, the method or field is simply skipped. An example:

```
@JvGroup (groups={"test"})
```

This example tells the framework that validation may only be applied for a group called test. The annotation has the following properties:

groups=The groups where validation of this method is applied for, it defaults to the *group* "1". Note that this property is a String array, thus you can specify multiple groups, e.g: {"group1","group2"} etc.

disableRecursionForGroups=Specify the groups where no validation must be applied (even if we would recurse). Defaults to none. For instance if the validator currently validates group "1" having a recursion of 2 deep, then if on a method this property is encountered and says "1" as well, the return value of that method is not validated (that would be the recursion). You will not often use this property, but it might be useful in certain circumstances (inheritance for instance).

`exposeInSubClass=true` | *false*. Use this property to *enable* annotations declared in baseclass A, be applied in child class B on the same method if it specifies a JvGroup with the same group name(s). This might sound hard, but its not that hard.

If you have a class A where a method is annotated with `@JvGroup(groups={"1"},exposeInSubClass=true)`. Now we have a class B extends A, where the same method is annotated (actually this method overrides A's method). Now B declares the following on that method: `@JvGroup(groups={"1"})`. Assume we are validating an instance of A first, nothing special happens - ordinary validation for group 1 is applied. Now assume we are validating an instance of B, the method of B is always validated for group 1, but what will happen with the one in A or better said the annotations? As `exposeInSubClass=true`, all **validation** annotations on that method will be applied too (assuming they fall in group 1).

So summarized: By using `exposeInSubClass=true` you can either enable or disable inheritance validation for a method.

Note: You *must* specify `@JvGroup` on each get method or field you need validation for. Not doing so, will skip the method /field for validation entirely.

## 4.1.6 @JvMethod

This is a supportive annotation that is used to specify a method; it is for example used in the annotations `@BeanLookup` and `@SpringLookup`.

```
beanLookup=@BeanLookup(  
    beanClass=BeanLookupValidation.class,  
    method=@JvMethod(  
        name="validate",  
        params={  
            @JvParam(  
                valueRetrievalMode=JvParam.MODE_CURRENT_OBJECT  
            ),  
            @JvParam(  
                valueRetrievalMode=JvParam.MODE_CURRENT_PATH  
            )  
        }  
    )  
)
```

This example specifies the method to use for a `BeanLookup`. The method has the following properties:

**name**=The name of the method. This should be the full name of the method you wish to use to perform the actual validation (or call something in other uses such as `@LovConstraint`).

**params**=Array of parameters. The parameters of the method, these are entirely up to you. This is an array of `@JvParam` annotations.

There is one catch though, your method must return whatever is specified in the constraints of annotations you use. For instance for validation (in `ValidateDefinition`), it should return a `List<ValidationMessage>`. But this requirement differs per type of use, so read well.

## 4.1.7 @JvParam

This annotation is used to specify a parameter (or more) for a `@JvMethod`. Parameters can be specified with literal values, but values can be looked up as well. An example:

```

method=@JvMethod(
    name="validate",
    params={
        @JvParam(valueRetrievalMode=JvParam.MODE_CURRENT_OBJECT),
        @JvParam(valueRetrievalMode=JvParam.MODE_CURRENT_PATH),

        @JvParam(valueRetrievalMode=JvParam.MODE_LOOKUP_SPRING, type="java.lang.Long", lookupName="SimpleServiceBean", requiresMethodCall=true, methodName="getLongValue")
    }
)

```

The parameters of this method contain a couple of special types as well, including a lookup parameter (getLongValue).

The annotation has the following properties:

valueRetrievalMode= *MODE\_VALUE* | *MODE\_BEAN* | *MODE\_LOOKUP\_SPRING* | *MODE\_CURRENT\_OBJECT* | *MODE\_CURRENT\_PATH*. Represents the type of parameter:

*MODE\_VALUE* should be used for direct input. You must specify the value directly using value and type properties.

*MODE\_BEAN* should be used if you wish to use a java object as input for this parameter; optionally a `getMethod` can be called on the bean to use the return value as input instead. You must specify the `beanClass` and `methodName` and preferably the type if known, optionally you can specify: `requiresMethodCall=true` and the 'get' `methodName` to call on the bean.

*MODE\_LOOKUP\_SPRING* should be used if you wish the parameter's value to be looked up through Spring. You must specify `lookupName` and optionally `requiresMethodCall=true` and a `methodName` if a `getMethod` must be called on the looked up instance.

*MODE\_CURRENT\_OBJECT* is a special parameter, by setting this you tell the framework that this parameter is the actual object currently under validation (thus must be passed as value for this parameter). You don't need to specify anything else.

*MODE\_CURRENT\_PATH* is a special parameter, by setting this you tell the framework that this parameter should be the current 'validation path' (thus must be passed as value for this parameter). For instance assume we are validating `Person.getName()` the path could be "" (or "person" depending on prefixes. The method performing validation can use this path to create proper `ValidationMessage` instances.

value=Set the actual value for this parameter. Note that it is only valid if `valueRetrievalMode` is *MODE\_VALUE*.

type=The type of the parameter, specify when *MODE\_VALUE* is also specified. For primitive values use their primitive type, e.g. `int`, `double`, `float` etc. Otherwise specify a fully classified name, e.g. `java.lang.Long`, `java.lang.String` etc.

lookupName=The name of the bean or whatever you wish to lookup.

requiresMethodCall=`true` | `false`. If you use *MODE\_BEAN* or *MODE\_LOOKUP\_SPRING* you can optionally specify a method call for that bean, the value returned will be used as input parameter then. Set this property to `true` then.

methodName=The name of the method to call on a bean / lookup. Must be an ordinary `getMethod()` name. E.g. `getMySpecialValue`. Only used if `requiresMethodCall=true`.

beanClass=Class of the bean to use when *MODE\_BEAN*.

As you can see the `@JvParam` is a powerful annotation if used properly.

## 4.2 Validation annotations

In the previous paragraph we have discussed the meta-related annotations that are used to denote that validation must be applied to objects, what groups and where to get data from. Here we look to the actual validation annotations.

The framework ships with a number of default validation annotations, we will discuss here how to use them. Extensions that provide more annotations are discussed in their respective chapter.

Each annotation has a validator, which implements the actual validation. When errors are encountered `ValidationMessage` instances are created, which contain an error code (from the default properties file that ships in the `jvalid-core.jar` file).

So by default the message for the codes from the property file is used, however each annotation we will discuss allows you to specify a custom message in case where the default message is not sufficient (or where you need another message than the one from the default property file or your own property file). See also 5.5 for customizing the default properties.

Beside specific property messages, you can enable global messages for each validation annotation if you need it in a special case. This allows you that certain messages will be converted to global messages (e.g. in JSF and Spring).

If you need to add custom annotations for validation, you can easily do that yourself, please check out 3.1 and 4.3.

Remember that the method you wish to annotate must have a `@JvGroup` annotation or all other validation annotations are ignored by default (no validation is applied).

### 4.2.1 Plural annotations

Since 1.1 the framework ships with plural annotations in addition to the normal annotations. Plural annotations are not validation annotations themselves, but allow to specify multiple validation annotations of the same kind on one method/field. For instance consider the following code:

```
@JvGroup (groups={"edit","advanced"})
@MinLengths (
    values={
        @MinLength(length=3,applyToGroups={"edit"}),
        @MinLength(length=10,applyToGroups={"advanced"})
    }
)
public String getName() { .. }
```

The plural annotation `@MinLengths` allows to add more than one `@MinLength` annotation. This is useful if different rules must be applied in certain cases of the same validation annotation. You can enforce it by the `applyToGroups` property as this example shows.

ALL validation annotations have a plural annotation (simply add an 's' to them and you found them!). The only ones *not* supporting a plural form are `@NotNull` and `@NotEmpty`.

### 4.2.2 @NotNull

Use this annotation to specify the method must return a non-null value. This is a special annotation, just like the `@NotEmpty` one. Other validations are ONLY applied if this one is valid (thus the value is indeed non-null). After all, other validations are useless if the value is null.

Important: Use either `@NotNull` or `@NotEmpty` annotation, but not both.

applyToGroups=Defaults to *JvGroup.GROUP\_APPLY\_ALL*. Use this property to specify 1 or more groups, when this check must be applied. This is useful if you have multiple groups specified in `@JVgroup`, this way you can turn on/off validation depending on the group you are currently validating through the validator. If not specified, this annotation is applied for any group specified in `@JVGroup`.

customCode=Defaults to empty string. If you need a specialized message for the validation error of this annotation, set it to a custom property (from one of your own property files).

globalMessage=Defaults to false. When set to true the annotated field/method that violates current constraint, will add a `ValidationMessage` with a global property set. This global property is used by the message converters that ship with the distribution. In general you do not want to set this to true, only in special cases where you require a global message instead of message for a validation path.

### 4.2.3 @NotEmpty

Use this annotation to specify the method must return a non-empty `String` value. This method can be specified on methods returning `java.lang.String` or `java.lang.StringBuffer`. This one is especially useful as certain techniques as JSF always fill a `String` with "" while the user originally left it empty in a web form.

Important: Use either `@NotNull` or `@NotEmpty` annotation, but not both.

applyToGroups=See definition in 4.2.2.

trim=*true* | false. Whether the `String` should be trimmed first, before applying the validation. If for example a method returns "" and trim is true, the `String` is considered empty – thus it is a validation error. If false, "" would be seen as valid (no error).

customCode=Defaults to empty string. If you need a specialized message for the validation error of this annotation, set it to a custom property (from one of your own property files).

globalMessage=See definition in 4.2.2.

### 4.2.4 @BetweenLength

Use this annotation to specify the method must return a `java.lang.String` or `java.lang.StringBuffer` value where the length must be between given constraints.

applyToGroups=See definition in 4.2.2.

minimumLength=Minimum length (inclusive) of the `String` / `StringBuffer`.

maximumLength=The maximum length (inclusive) of the `String` / `StringBuffer`

customCode=Defaults to empty string. If you need a specialized message for the validation error of this annotation, set it to a custom property (from one of your own property files).

globalMessage=See definition in 4.2.2.

### 4.2.5 @MaxLength

Use this annotation to specify the method must return a `java.lang.String` or `java.lang.StringBuffer` value where the length must not be longer than given maximum length.

applyToGroups=See definition in 4.2.2.

length=The maximum length allowed (inclusive) of the String / StringBuffer.

customCode=Defaults to empty string. If you need a specialized message for the validation error of this annotation, set it to a custom property (from one of your own property files).

globalMessage=See definition in 4.2.2.

## 4.2.6 @MinLength

Use this annotation to specify the method must return a java.lang.String or java.lang.StringBuffer value where the length must not be less than given minimum length.

applyToGroups=See definition in 4.2.2.

length=The minimum length (inclusive) the String / StringBuffer's length must be.

customCode=Defaults to empty string. If you need a specialized message for the validation error of this annotation, set it to a custom property (from one of your own property files).

globalMessage=See definition in 4.2.2.

## 4.2.7 @MinValue

Use this annotation to specify the minimum return value of this method (or field value). Can be specified on methods or fields with type:

- java.lang.Byte
- java.lang.Short
- java.lang.Integer
- java.lang.Long
- java.lang.Float
- java.lang.Double

As well as methods/fields with the equivalent primitive type of this list. In this case use the value() property of the annotation. This annotation also supports:

- java.math.BigInteger
- java.math.BigDecimal

If you annotate it on BigInteger / BigDecimal you must use the bigValue() property instead.

applyToGroups=See definition in 4.2.2.

value=The minimum value. Defaults to 0.0.

bigValue=The minimum value for BigInteger/BigDecimal. Defaults to an empty String.

customCodeMinValue=Defaults to empty string. If you need a specialized message for the minimum validation error of this annotation, set it to a custom property (from one of your own property files).

customCodeNotNumeric=Defaults to empty string. If you need a specialized message for the not-numeric validation error of this annotation, set it to a custom property (from one of your own property files).

globalMessage=See definition in 4.2.2.

Note: Double and/or float values might not produce results you always expect! Be cautious for even using these types. You'd better use BigDecimal in your model in that case.

## 4.2.8 @MaxValue

Use this annotation to specify the maximum return value of this method (or field value). Can be specified on methods or fields with type:

- java.lang.Byte
- java.lang.Short
- java.lang.Integer
- java.lang.Long
- java.lang.Float
- java.lang.Double

As well as methods/fields with the equivalent primitive type of this list. In this case use the `value()` property of the annotation. This annotation also supports:

- java.math.BigInteger
- java.math.BigDecimal

If you annotate it on BigInteger / BigDecimal you must use the `bigValue()` property instead.

applyToGroups=See definition in 4.2.2.

value=The maximum value. Defaults to 0.0.

bigValue=The maximum value for BigInteger/BigDecimal. Defaults to an empty String.

customCodeMinValue=Defaults to empty string. If you need a specialized message for the minimum validation error of this annotation, set it to a custom property (from one of your own property files).

customCodeNotNumeric=Defaults to empty string. If you need a specialized message for the not-numeric validation error of this annotation, set it to a custom property (from one of your own property files).

globalMessage=See definition in 4.2.2.

Note: Double and/or float values might not produce results you always expect! Be cautious for even using these types. You'd better use BigDecimal in your model in that case.

## 4.2.9 @LovConstraint

Use this annotation to specify that a value (that a method returns) is either *in* or *not in* a list of values. This is a powerful annotation, as it supports both literal values specified in the annotation itself but as well lets you lookup list of values through the use of Javabeans or Spring (thus in a dynamic way).

The annotation supports List of values (lov) that must be an array or anything that implements `java.util.Collection`. Thus for dynamic lookup your values must be either an array or a class implementing `java.util.Collection`.

applyToGroups= See definition in 4.2.2.

operator= `OPERATOR_IN` | `OPERATOR_NOT_IN`

The operator to use when validating a methods return value. If `OPERATOR_IN` is specified, the value must exist inside the array or collection. If `OPERATOR_NOT_IN` is used the value may not exist in the array or collection. If the value violates one of these rules, a validation error occurs.

values=The value array if you need to specify values directly. This is a `java.lang.String[]` array. It can be used to specify primitives as well. Supported types are:

- java.lang.Byte / byte
- java.lang.Short / short
- java.lang.Integer / int
- java.lang.Long / long
- java.lang.Float / float
- java.lang.Double / double
- java.lang.String

Just specify your values as `String`, and set the `type` property to one of the items listed above. Make sure the values you specify here can be converted to given type. If you specify this property type must be specified as well.

**type**=The type of the `String`'s in the `value` property array (if you use the `values` property). This one should be the correct one you had in mind, and match the method's return value that is annotated by this annotation. Types supported are mentioned above.

**dynamic**=Whether this `LovConstraint` is dynamic or not. If `false` you should use the `values` / `type` properties. If `true` you must specify the `lookup` property. You should use `dynamic=true` if you have a dynamic collection / array as input for this `LovConstraint`.

**lookup**=The lookup to use to get a list or array. Using this `@Lookup` you can specify to use a Javabean or Spring bean. See `@Lookup` for info on how to specify this. Important: Your bean, must return either `java.util.Collection` or an array (array can be anything, but should be of the same objects normally to have any sane comparison).

**globalMessage**=See definition in 4.2.2.

A few examples to clarify things:

```
@JvGroup
@LovConstraint (values={"a","b","c","d","e"})
public String getStrValue() {
    return strValue;
}
```

This example defines that the method `getStrValue()` must return a `String` value that must be in the `LovConstraint` (thus one of the values array). If not a validation error occurs.

```
@JvGroup
@LovConstraint (type="int",values={"1","2","3","4","5"})
public int getIntValue() {
    return intValue;
}
```

The second example defines that the method `getIntValue()` must return a value that is in given `LovConstraint`, note that the type specified is 'int' (as the values in the array must be converted to that first before comparison).

```
@JvGroup
@LovConstraint (
    dynamic=true,operator=LovConstraint.OPERATOR_IN,
    lookup=@Lookup(
        type=Lookup.LOOKUP_NORMAL_BEAN,
        beanLookup=@BeanLookup(
            beanClass=LovService.class,
            method=@JvMethod(name="getShortValues")
        )
    )
)
public Short getShortValue() {
    return shortValue;
}
```

The last example shows that the `LovConstraint` uses dynamic input, in this example it uses a Javabean by using `@BeanLookup`. `BeanLookup` creates a class named `LovService` and calls a method on that instance named 'getShortValues', this method thus must return a collection or an array containing the allowed `Short` values (to have a useful check that is).

customCodeLovIn=Defaults to empty string. If you need a specialized message for the 'not-in' validation error of this annotation, set it to a custom property (from one of your own property files).

customCodeLovNotIn= Defaults to empty string. If you need a specialized message for the 'in' validation error of this annotation, set it to a custom property (from one of your own property files).

Summarized you can use this annotation to compare even a single value (your array / collection will only contain 1 value then!), write the values directly using the values array or lookup the array / collection dynamically.

#### **4.2.10 @RegularExpression**

Use this annotation to specify a regular expression the annotated method's value must match. Can be specified on method's returning a java.lang.String or java.lang.StringBuffer.

pattern=The regular expression pattern the value must comply to. See java.util.regex.Pattern for more information on how to write regular expressions. Required.

applyToGroups= See definition in 4.2.2.

customCode=Defaults to empty string. If you need a specialized message for the validation error of this annotation, set it to a custom property (from one of your own property files).

globalMessage=See definition in 4.2.2.

#### **4.2.11 @ValidateList**

Use this annotation if you want a List (or any implementation of java.util.List) with objects to be validated, each object is validated in the list (assuming that these objects have a @ValidateDefinion defined).

applyToGroups= See definition in 4.2.2.

globalMessage=See definition in 4.2.2.

Note: Automatic validation in JSF may or may not work (related to automatically creating/finding the components and creating JSF messages), depending on how you have created your input components, however you can use the shipping validate tag in that case to make it work.

#### **4.2.12 @ValidateMap**

Use this annotation if you want a Map with objects to be validated, each object is validated in the Map (assuming that these objects have a @ValidateDefinion defined).

applyToGroups= See definition in 4.2.2.

globalMessage=See definition in 4.2.2.

Note: Automatic validation in JSF may or may not work (related to automatically creating/finding the components and creating JSF messages), depending on how you have created your input components, however you can use the shipping validate tag in that case to make it work.

#### **4.2.13 @CollectionSize**

Use this annotation to check the size of a `java.util.Collection` or `java.util.Map` (and any related classes). The `CollectionSize` annotation offers different modes to perform the following type of checks:

- `MODE_EQUALS`
- `MODE_NOT_EQUALS`
- `MODE_MINIMUM`
- `MODE_MAXIMUM`
- `MODE_BETWEEN`

Each mode performs a check related to the collection's/map's size. The default mode is `MODE_EQUALS`. When `MODE_EQUALS` is used the collection's size must match the `size()` property on the annotation.

With `MODE_NOT_EQUALS` the collection's size must *not* match the `size()` property of the annotation.

When using `MODE_MINIMUM` the collection's size must be at least the `minimumSize()` of the annotation.

With `MODE_MAXIMUM` the collection's size must be at maximum the `maximumSize()` of the annotation.

Using the `MODE_BETWEEN`, the collection's size must be between the `minimumSize()` and `maximumSize()` properties of the annotation.

The annotation has the following properties:

mode=The mode of the check, use one of the above discussed modes (they are constants on the annotation). Defaults to `MODE_EQUALS`.

size=The size for the collection/map (depends on mode, relevant only for `MODE_EQUALS` / `MODE_NOT_EQUALS`). Defaults to 1.

minimumSize=The minimum size for the collection/map (relevant only for `MODE_MINIMUM` and `MODE_BETWEEN`). Defaults to 1.

maximumSize=The maximum size for the collection/map (relevant only for `MODE_MAXIMUM` and `MODE_MAXIMUM`). Defaults to 1.

customCode=Defaults to empty string. If you need a specialized message for the validation error of this annotation, set it to a custom property (from one of your own property files).

applyToGroups= See definition in 4.2.2.

globalMessage=See definition in 4.2.2.

Note: Each mode has its own message code property for validation message's (see `MessageCodes` class or check the `jav_messages.properties` that ships with the distribution).

## 4.2.14 @DateCheck

This annotation allows you to check a date with the following options (as modes):

- `MODE_EQUALS`
- `MODE_NOT_EQUALS`
- `MODE_LESS_THAN`
- `MODE_MORE_THAN`
- `MODE_EQUALS_LESS_THAN`
- `MODE_EQUALS_MORE_THAN`

By default the annotation's mode is `MODE_EQUALS`. The following properties are available on the annotation:

today=If date of today must be used as check, defaults to true. If you want to check against another date you must set it to false.

mode=The mode of the check this annotation will do. Use one of the constants mentioned above. It defaults to `MODE_EQUALS`.

convertType=The dates are converted to this type, meaning that the actual comparison is done this way (e.g TYPE\_CONVERT\_DATE means that the dates are compared on their date's only, time is not included). Defaults to TYPE\_CONVERT\_DATE. In fact it is the accuracy of the comparison. Possible values are:

- TYPE\_CONVERT\_DATE (comparison is done on day,month and year)
- TYPE\_CONVERT\_FULL (comparison is done on day,month,year, hour,minute and seconds)
- TYPE\_CONVERT\_MILLIS (comparison is done in milliseconds exactly)

value=The date to use as a check, only valid if the property today is set to false. You must specify a valid date here in the format dictated by the pattern() property (see below). Defaults to an empty string.

pattern=The pattern to use to convert the value() property to a valid date. For formats supported see java.util.SimpleDateFormat. This property defaults to yyyy-MM-dd.

customCode=Defaults to empty string. If you need a specialized message for the validation error of this annotation, set it to a custom property (from one of your own property files).

applyToGroups= See definition in 4.2.2.

globalMessage=See definition in 4.2.2.

So summarized if you just specify @DateCheck it will check the date annotated if it equals today's date. To change this behavior use the properties above.

## 4.3 Add custom validation annotations

In this chapter a lot of annotations were discussed. As seen the framework ships with a number of validation annotations. You can use these for your applications, however often you might need a specialized validation not provided by the framework (yet).

It is not hard to add a new custom validation annotation. To add a new annotation you must do the following:

- Create your annotation (make sure it can be applied on methods and fields)
- Create a class that implements org.javalid.core.validator.JavalidValidator
- Register your new annotation and validator in the configuration xml file of JaValid

As an example the @MinLength annotation is used, you can do the same for your own annotation.

### 4.3.1 Create the annotation

```
@Target (value={ElementType.METHOD,ElementType.FIELD})
@Retention (RetentionPolicy.RUNTIME)
@Documented
public @interface MinLength {

    long length();

    String[] applyToGroups() default { JvGroup.GROUP_APPLY_ALL};

    String customCode() default "";

    boolean globalMessage() default false;
}
```

The code below (stripped off its comments) defines the MinLength annotation. What you put in your annotation is entirely up to you. The applyToGroups() property is recommended though, as is the globalMessage property.

## 4.3.2 Create the validator implementing JavalidValidator

```
public class MinLengthValidatorImpl implements JavalidValidator<MinLength> {

    private ValidatorSupport validatorSupport;

    public MinLengthValidatorImpl() {
        validatorSupport = new ValidatorSupport();
    }

    public List<ValidationMessage> validate(MinLength annotation, Object value, String path, JvConfigurationWrapper config)
    {
        String strValue = null;
        if(value instanceof String) {
            strValue = (String)value;
        }
        else
        if(value instanceof StringBuffer) {
            strValue = ((StringBuffer)value).toString();
        }
        else {
            throw new JavalidException("Given value is not of type java.lang.String or java.lang.StringBuffer");
        }

        List<ValidationMessage> list = new ArrayList<ValidationMessage>();
        if(strValue.length() < annotation.length()) {
            String messageCode = validatorSupport.isEmptyString(annotation.customCode()) ? MessageCodes.MSG_MIN_LENGTH_ERROR :
            annotation.customCode();

            list.add(new ValidationMessage(path,messageCode,new Object[]
            { value,annotation.length() },annotation.globalMessage()));
        }
        return list;
    }

    public boolean validationMustBeAppliedToGroup(MinLength annotation,
        String groupName,
        JvGroup jvGroup,
        JvConfigurationWrapper config) {
        return validatorSupport.validationMustBeAppliedToGroup(annotation.applyToGroups(),groupName,jvGroup.groups());
    }
}
```

The JavalidValidator interface defines two methods:

- public List<ValidationMessage> validate(T annotation, Object value, String path, JvConfigurationWrapper config)
- public boolean validationMustBeAppliedToGroup(..)

The first method is called by the framework to validate given value associated with given annotation (here MinLength but for your own constraint this would be your annotation). The second method is called by the framework *before* calling validate, to determine it must validate for given group.

Note: Your class must have a public non-args constructor.

## 4.3.3 Add your annotation to the configuration

The JaValid configuration file has a section with <annotation> elements where you can register your annotation. Simply add a new entry like this:

```
<annotations>
....
    <annotation annotation-class="org.javalid.annotations.validation.MinLength"
        validator-class="org.javalid.core.validator.MinLengthValidatorImpl" />
....
</annotations>
```

That's all there is to add any custom type of annotation to JaValid. Since 1.1 you can also add a plural annotation for your annotation as discussed below.

### 4.3.4 Plural annotations

In the previous sections we discussed how to add a new annotation and register it. Since JaValid 1.1 you can also add a plural form of your annotation. This means that you can annotate a method/field with multiple of the same annotations. To do so you must add a new annotation which must adhere the following rules:

- It can be applied on method / fields
- It has a values() array property, which allows you to add multiple annotations

An example: The previously discussed @MinLength annotation has a plural form too, the code looks like this:

```
@Target (value={ElementType.METHOD,ElementType.FIELD})
@Retention (RetentionPolicy.RUNTIME)
@Documented
public @interface MinLengths {

    MinLength[] values();

}
```

As you can see we added an s to the original name (this is not required, but recommended as it makes configuration shorter). We also added a values() element which is an array of the MinLength annotation. This allows you to write: @MinLengths(values={@MinLength(...),@MinLength(...)}). You should do the same for your own annotation.

The final step is to add the plural annotation to the configuration so JaValid knows it is there. For @MinLength it would look like:

```
<annotation annotation-class="org.javalid.annotations.validation.MinLength"
            validator-class="org.javalid.core.validator.MinLengthValidatorImpl"
            supports-plural="true" />
```

The only thing added is the supports-plural property. If set, the framework will automatically detect the plural annotation (by adding the s to the normal annotations name!). There is nothing else you need to do. If supports-plural is omitted it will automatically default to false (so no plural annotation will be supported in that case).

In some cases you may want to name your plural annotation differently that with just an s appended, in that case you would have to specify it like this in the configuration:

```
<annotation annotation-class="org.javalid.annotations.validation.MinLength"
            validator-class="org.javalid.core.validator.MinLengthValidatorImpl"
            supports-plural="true"
            annotation-class-plural="test.MyPluralAnnotation" />
```

You need to add an extra attribute 'annotation-class-plural' and specify the plural annotation.

## 4.4 Annotation value resolver

This is a special class which ships within the core distribution. The full classname is: **org.javalid.core.AnnotationValueResolver**

### 4.4.1 Introduction

This class is written and added to make things easier in the front-end of a web application. In many cases you will annotate your business objects with constraints (e.g. @MaxLength, @MinValue among a few). The annotations have properties themselves, which are only 'exposed' when a validation error occurs (e.g. the user submits a String which is too short). You can notify the user about his or her error in that case.

Often you have input controls such as textfields e.g. where the user can provide input. It would be convenient that if you have for instance a businessobject/method Person.firstName annotated with @MaxLength (value=20) that your textfield would indeed not accept more than 20 characters in the frontend of the application. Normally you would have to write it yourself in the page, with the risk if ever this constraint changes you need to change the page too. These are things that are often forgotten.

The resolver allows you to retrieve annotation values to the frontend of your application, thus you effectively bind certain frontend properties to an annotation value, so if the annotation's value ever changes it is automatically reflected to the frontend.

An example to clarify things, in this example we use JSF, but it can be used in normal JSP too for instance through jstl (though you'd probably use the requestScope to expose the resolver).

```
@ValidateDefinition
public class Person {
    private String firstName;

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    @JvGroup (groups={"1"})
    @NotNull
    @MaxLength (value=30)
    public String getFirstName() {
        return this.firstName;
    }
}

public class PersonBean {

    public Person getPerson() {
        return this.person;
    }

    public AnnotationValueResolver getResolver() {
        return annotationValueResolver;
    }
}

<h:inputText value="#{PersonBean.person.firstName}"
maxLength="#{PersonBean.resolver['Person,MaxLength:value,m:firstName']}" />
```

The code above is divided in 3.

The first part declares a Person business object and it is annotated with some (validation) annotations. Note the @MaxLength constraint, defining the firstName may not be more than 30 characters long.

The second part is a JSF (session) bean, which exposes a Person instance that can be edited in a web page. Note that it also exposes the AnnotationValueResolver class for the sake of simplicity. You could also define that class as a bean itself or expose it differently – crucial is that it must be available for the page somehow.

Finally the third part is a fragment from a JSF page declaring an h:inputText component for user input, the value shows the user can update the firstName of person. Now have a close look on the maxLength property of the input component:

```
#{PersonBean.resolver['Person,MaxLength:value,m:firstName']}"
```

The bolded text shows what needs to be done to retrieve the value from the @MaxLength annotation declared on the getFirstName() method. As you can see you need to define a special key when you access the map, this key is then used to retrieve an annotation value.

One final note, you can retrieve nearly any value from an annotation, keep that in mind.

Now that we have seen an example on how to use the Resolver, let's discuss what needs to be done exactly to successfully use it.

## 4.4.2 Configuration

There is not much configuration needed as it currently works as a standalone class and has no direct associations with the framework (except for the annotations and support classes).

So basically you need only to create one instance of the class and share that in your application. However you must define aliases for your business objects in order to easily refer to them from your page when you write a key. This can be achieved by calling the method:

```
public void setAliasesForClassMap(Map<String,String> aliasesMap)
```

The aliasesMap parameter must contain aliases as keys for your business objects, and as values the fully qualified business object. For example: Person,org.javalid.example.Person would be a valid entry (Person is now the alias for the class). So basically if you need to have access to your business object later on you should define it here.

For usage with JaValid annotations there is nothing more needed than previous method call.

By default the following annotations are available for usage (first their alias, then the actual annotation class):

### Class level annotations:

ValidateDefinition / org.javalid.annotations.core.ValidateDefinition</li>

### Method level annotations:

JvGroup	org.javalid.annotations.core.JvGroup
BetweenLength	org.javalid.annotations.validation.BetweenLength
LovConstraint	org.javalid.annotations.validation.LovConstraint
MaxLength	org.javalid.annotations.validation.MaxLength
MinLength	org.javalid.annotations.validation.MinLength
MaxValue	org.javalid.annotations.validation.MaxValue
MinValue	org.javalid.annotations.validation.MinValue
NotEmpty	org.javalid.annotations.validation.NotEmpty
NotNull	org.javalid.annotations.validation.NotNull
RegularExpression	org.javalid.annotations.validation.RegularExpression
ValidateList	org.javalid.annotations.validation.ValidateList
ValidateMap	org.javalid.annotations.validation.ValidateMap
CollectionSize	org.javalid.annotations.validation.CollectionSize
DateCheck	org.javalid.annotations.validation.DateCheck

If you need **custom annotations** to be available (either JaValid related or anything else you can come up with), you can add them to the configuration of the class by calling:

**public void setRegisteredAnnotationsMap(Map<String,String> annotationsMap)**

The annotationsMap must contain as keys aliases for your annotations, and as values the fully qualified annotation class. For an example see the list above. That's it, now you can also read from any custom annotation you add!

One last optional configuration method:

**public void setCacheResults(boolean cacheResults)**

By default results resolved are NOT cached, it is recommended to cache however – by calling this method with a true parameter you assure that results are cached and not again resolved if the same key is requested.

When done configuring, your resolver should reside somewhere you have easy access to it. In JSF you could declare it as a bean for instance, the same applies to Spring.

### 4.4.3 Key usage

In the example in the introduction of this chapter a key was used to retrieve a value from an annotation. The key you pass in the value map, must adhere to the following formal syntax (comma separates parts):

AliasObjectToCheck,AnnotationAlias:method,c|(m:method)

- The first part is the object's alias you need to read the annotation from (in our example Person).
- The second part is the alias of the annotation you want to read followed by a semicolon after which the full method call must follow (the method to be called on the annotation). You can nest calls by separating with a dot. For example: LovConstraint:lookup.beanLookup.method.name would return the name of a JvMethod inside the beanLookup. Methods may not have parameters.
- The third part tells where the annotation is found. The first part tells which object the annotation should reside, this part tells if method traversal is needed first. There are two options currently: One where we need to find the annotation on class level, and one on method level (field is not supported for now).
  - Class level is simply denoted by a c. Nothing should follow.
  - Method level is denoted by a m followed by semicolon and then the method (or more nested methods) to locate and from there the annotation is taken. Note that these methods use expression syntax so getValue() you should write as value.

Example of both:

- Person,ValidateDefinition:primaryGroup,c (returns the primaryGroup of @ValidateDefinition that is annotated on the Person's class)
- Person,MaxValue:value,m:someValue (returns the value of the @MaxValue annotation that is annotated on the Person's class getSomeValue() method).

Nested example:

- LovConstraintLookupObject,LovConstraint:lookup.beanLookup.method.name,m:shortValue (returns the methodname of @JvMethod which is encapsulated by @BeanLookup, @Lookup and @LovConstraint respectively).

You can use the resolver for any annotation you wish to read from!

## 5 Validator usage

This chapter describes the general use of the validator. You will learn how to programmatically use the framework. For information on using the validator with the JSF (Java Server Faces) framework please see the next chapter.

In the previous chapters you have learned about the general configuration and annotation usage of the framework. Now we will have a look at the actual use of the validator.

### 5.1 Interface

The validator is defined in the interface: **org.javalid.core.AnnotationValidator**, its complete definition is shown below (without all comments from the original file to keep it brief).

```
package org.javalid.core;

import java.util.List;

public interface AnnotationValidator<T> {

    public List<ValidationMessage> validateObject(T object);

    public List<ValidationMessage> validateObject(T object, String group);

    public List<ValidationMessage> validateObject(T object, String group,
        JavalidValidationCallbackHandler callbackHandler);

    public List<ValidationMessage> validateObject(T object, String group,
        boolean recurse);

    public List<ValidationMessage> validateObject(T object, String group,
        boolean recurse,
        JavalidValidationCallbackHandler callbackHandler);

    public List<ValidationMessage> validateObject(T object, String group,
        String pathPrefix);

    public List<ValidationMessage> validateObject(T object, String group,
        String pathPrefix,
        JavalidValidationCallbackHandler callbackHandler);

    public List<ValidationMessage> validateObject(T object, String group,
        String pathPrefix,
        boolean recurse);

    public List<ValidationMessage> validateObject(T object, String group,
        String pathPrefix,
        boolean recurse,
        JavalidValidationCallbackHandler callbackHandler);

    public List<ValidationMessage> validateObject(T object, String group,
        String pathPrefix,
        boolean recurse,
        int levelDeep);

    public List<ValidationMessage> validateObject(T object, String group,
        String pathPrefix,
        boolean recurse, int levelDeep,
        JavalidValidationCallbackHandler callbackHandler);

    public List<ValidationMessage> validateProperty(T owningObject,
        String groupName,
        Object propertyValue,
        String propertyName,
        String pathPrefix);
}
```

It is this interface you will use either directly or indirectly for validating your objects. There are two distinct methods:

- `validateObject(..)`
- `validateProperty(..)`

The first is used for validating an object (thus all its properties that require validation), the second one is used for validation of a single property. You will normally not use `validateProperty()` directly, it is for use of a special `JaValid` validator tag in JSF (which can be found in the JSF extension `.jar` file).

The other methods simply add a couple of extra parameters, nothing more. In most cases you can use the simplest version of the interface: **`validateObject(T object, String group)`**. The method expects the object you wish to validate, and what group. That's it! You can use other methods for more control, which might be needed in some cases.

For instance **`validateObject(T object, String group, String pathPrefix)`** allows you to provide a prefix for the validation path. Normally it starts as `""`, you might need this where you have a path already (e.g. in JSF where your object is usually bound like: `backingBean.myObject`, you might want to add the prefix path `backingBean`). This assures that the validation path in the messages is correct.

The `JaValidValidationCallbackHandler` parameter is new since 1.1 and is discussed in paragraph 5.3.

## 5.2 Implementation

In the previous paragraph we had a look at the interface of the validator. Now we will discuss the available implementations.

The core framework ships with one implementation: **`org.javalid.core.AnnotationValidatorImpl`**. In most cases you will do nothing with this class directly, however if you programmatically need to instantiate the framework you will touch it one time constructing it.

The class has four constructors:

```
public AnnotationValidatorImpl()

public AnnotationValidatorImpl(String xmlConfigFile)

public AnnotationValidatorImpl(String xmlConfigFile, boolean
setValueToNullIfEmptyString)

public AnnotationValidatorImpl(String xmlConfigFile, boolean
setValueToNullIfEmptyString, boolean callConfigurationLoadedComplete)
```

What constructor you need to use, depends on your needs.

**`public AnnotationValidatorImpl()`** can be used to quickly initialize the framework where you only need core functionality. The framework will use the default configuration file that ships with it (it can be found in: `org/javalid/core/config/jv-config.xml`). This will initialize the framework with core annotations available, lookups are disabled.

**`public AnnotationValidatorImpl(String xmlConfigFile)`** can be used to provide your own configuration file. The framework will then use this configuration, the default configuration is NOT loaded so make sure you copy the original configuration file to your own file and make your changes then.

**`public AnnotationValidatorImpl(String xmlConfigFile, boolean setValueToNullIfEmptyString)`** is a specialized version of the previous constructor and can be used to override default behaviour of an empty string (applies to `java.lang.String` and `java.lang.StringBuffer`). If `setValueToNullIfEmptyString` is set to true, an empty value `""` is set to null (assuring no other validations are fired). This parameter was originally used in combination

with the `@NotNull` validation annotation, you can however also use `@NotEmpty` instead and instead use the previous constructor. This parameter defaults to false.

**`public AnnotationValidatorImpl(String xmlConfigFile, boolean setValueToNullIfEmptyString, boolean callConfigurationLoadedComplete)`** is new since 1.1 and introduces an extra parameter which denotes if a call to extensions must be made once loading of this constructor is complete, defaults to true (for all other constructors). In some cases it is necessary to do the call on another time, e.g. `SpringAnnotationValidatorImpl` does this as soon as the Spring context is available (would it do this earlier failures occur for instance with the database extension if that one defines a spring datasource).

An example to create a validator:

```
AnnotationValidator validator = new AnnotationValidator("mypackage/my-config.xml");
```

Yes, that is all there is to create your validator! This validator is ready for usage right away.

### 5.3 Callback handler

Since 1.1 JaValid allows you to pass a Callback handler when you call a `validateObject(..)` method, this is an optional parameter.

The handler allows you to get notifications from the framework during the validation. It allows you to change validation messages for instance if required. To do this a class must implement the `org.javalid.core.JavalidValidationCallbackHandler` interface. This interface defines the following methods (stripped of comments):

```
public interface JavalidValidationCallbackHandler<T> {  
  
    public void beforeValidation(T object, String currentGroup,  
        String currentValidationPath, List<ValidationMessage> messages);  
  
    public void afterValidation(T object, String currentGroup,  
        String currentValidationPath, List<ValidationMessage> messages);  
  
    public void beforeValidationRound(Object object, String currentGroup,  
        String currentValidationPath, int currentLevelDeep,  
        int maxLevelDeep, List<ValidationMessage> messages);  
  
    public void afterValidationRound(Object object, String currentGroup,  
        String currentValidationPath, int currentLevelDeep, int maxLevelDeep,  
        List<ValidationMessage> messages);  
  
}
```

The `beforeValidation` method is called *before* the framework starts the actual validation (so this is called only once). The object passed is the object you requested validation for. This method is called only once.

The `afterValidation` method is called *after* the validation is completed. The object passed is the object requested validation for. This method is called only once.

The `beforeValidationRound` is called during validation and is called for each object (and children if recursion is enabled) *before* it is validated (e.g. you validate `Order`, and it has a member called `Info`, the `Info` object will be passed here).

The `afterValidationRound` is called for each object (and children if recursion is enabled) *after* it is validated.

## 5.4 Exception handling

The framework is for validating your objects, but it's possible that an exception occurs. Leaving any not known bugs out, all exceptions are raised by the framework itself and will always be of type: **org.javalid.core.JavalidException**. This is a runtime exception; during normal and correct use of the framework this exception is *never* raised. The only exception is for JSF, a FacesException is thrown in that case (that is because JSF currently has bad exception handling and our exceptions are not handled properly).

This exception is only raised when incorrect usage is detected, such as annotations missing required properties, invalid values for properties, and the xml config file which cannot be loaded and so on. In your application you are advised that this type of exception is logged, so you can take appropriate action. Do not try to catch it yourself, as it normally means something is wrong on how you use the framework! The framework tries to raise a sensible exception every time, so check out its message on what might be wrong.

## 5.5 Validation Messages (i18n)

During validation, errors on the business object data might be found, these are stored inside ValidationMessage instances as said earlier.

The error codes used by the core validation are taken from org.javalid.core.validator.MessageCodes (the key definitions that is). The framework ships with a properties file, which you can use as your basis for looking up the actual text belonging to a code. You can copy the content of the property file to your own property file(s). The property file of the framework is: **org.javalid.core.validator.jv\_messages.properties**.

You usually need to worry about this only when you convert the ValidationMessage list returned by the AnnotationValidator to another type of message (e.g. SpringMessage, FacesMessage ..). Converters supplied by the framework simply ask you for the bundle to use then.

Other than changing the default messages, all validation annotations offer you a way to customize an error message in addition (thus you can have default messages and a customized message where needed).

## 5.6 Singleton, threading ...

This title is not really appropriate, but it catches your attention anyway now doesn't it? The framework is not a singleton framework; however its usage feels like a singleton.

The AnnotationValidator we learned to initialize earlier can be used as a single entry in a multithreaded environment. Thus it can be accessed by multiple threads and you need nothing to do for that.

Therefore do not initialize the Validator each time you need validation, which takes unnecessary time (loading a configuration file etc). You are recommended to share it somewhere for instance in a class that grants 'singleton' access. Here is a simple example to get you started.

```
package example;

import org.javalid.core.AnnotationValidator;
import org.javalid.core.AnnotationValidatorImpl;

public class ValidatorEntry {

    private static AnnotationValidator validator = null;

    private ValidatorEntry() {
    }

    public static final AnnotationValidator getValidator() {
        if(validator == null) {
            validator = new AnnotationValidatorImpl();
        }
        return validator;
    }
}
```

There are obviously more ways to achieve this; you can store it in a shared application context (e.g. ServletContext in a webapplication) for instance as well. In a framework like Spring you can define the Validator as a bean and inject it where you need it and so on.

You can even have different configured validators living together in the same application.

## 6 Spring extension

The framework provides a simple extension for Spring in a separate jar file. This extension allows you to use Spring beans in cases where complex validation is required.

### 6.1 Configuration and usage

1. First you need to add the javalid-spring-dep.jar library in your application's classpath.
2. The second step is to enable the Spring extension (changed since 1.1):

```
<extension name="spring" extension-loader-  
class="org.javalid.external.spring.extension.JavalidExtensionSpringImpl">  
  <parameters>  
    <parameter name="validatorClass"  
value="org.javalid.external.spring.validator.SpringValidatorExtImpl" />  
    <parameter name="lookupClass"  
value="org.javalid.external.spring.lookup.SpringLookupExtImpl" />  
  </parameters>  
</extension>
```

As you can see the spring configuration uses two classes (two parameters). The SpringValidatorImpl implements the following interface (documentation is left out for the sake of clarity).

```
package org.javalid.core.extvalidator;  
  
import java.util.List;  
  
import org.javalid.annotations.core.JvMethod;  
import org.javalid.core.ValidationMessage;  
import org.javalid.core.config.JvConfiguration;  
import org.javalid.core.extlookup.SpringLookupExt;  
import org.javalid.core.validator.JvConfigurationWrapper;  
  
public interface SpringValidatorExt {  
  
    public List<ValidationMessage> validate(Object currentInstance, String prefixPath,  
String beanName, JvMethod jvMethod);  
  
    public void setJvConfigurationWrapper(JvConfigurationWrapper jvConfigWrapper);  
  
    public void setSpringLookup(SpringLookupExt springLookupExt);  
  
}
```

The interface defines a couple of methods that are called by the framework. The validate(..) method is called if a Spring bean is needed to perform complex validation. The other two methods are called during configuration of the framework and provide access to the core configuration and the SpringLookupExt interface. You can provide your own implementation if needed.

The SpringLookupImpl implements the following interface (again documentation is left out for the sake of clarity).

```

package org.javalid.core.extlookup;

import org.javalid.core.validator.JvConfigurationWrapper;

public interface SpringLookupExt {

    public Object getSpringBean(String name);

    public void setJvConfigurationWrapper(JvConfigurationWrapper jvConfigWrapper);

    public void setApplicationContext(Object context);

}

```

This interface is used by the framework to lookup a Spring bean through the method `getSpringBean(String name)`. The other methods respectively set the configuration of the framework and set the applicationcontext of Spring (thus context should be an implementation of Spring's `ApplicationContext`). We use `Object` as parameter as this interface is part of the core framework and we don't want any direct dependencies with Spring there. You can provide your own implementation of this interface if needed.

3. The last step is to use a different `AnnotationValidator`, not the default one (as that one is not Spring aware). The extension provides **`org.javalid.external.spring.SpringAnnotationValidatorImpl`** which you can use right out of the box.

This validator extends `org.javalid.core.AnnotationValidatorImpl` and implements Spring's `org.springframework.context.ApplicationContextAware` interface.

The validator provides two constructors, one of them is **public `SpringAnnotationValidatorImpl(String xmlConfigFile)`**. As you can see you must provide your own configuration file, this is needed as the default configuration file is not Spring enabled. The second has the extra `setValueToNullIfEmptyString` parameter.

You can now easily define this validator in your Spring configuration, for instance like:

```

<bean id="SpringValidator" class="
org.javalid.external.spring.SpringAnnotationValidatorImpl">
  <constructor-arg index="0" value="org/javalid/springtest/config/jv-config.xml" />
</bean>

```

The validator is no different than the core implementation; the only difference is that it's Spring aware.

Note: If you implement your own classes of the interfaces discussed here, keep in mind that they must be able to support multiple threads simultaneously (just don't use member data that changes).

## 6.2 Spring Validator

In the previous paragraph we discussed how to use the `SpringAnnotationValidator` class. Besides using the validator directly in your Spring MVC beans (or any other spring bean) there is also an option to use a special Validator class.

The jar file ships with the **`org.javalid.external.spring.SpringValidator`** class. This class implements the Spring `org.springframework.validation.Validator` interface. You can inject this class into your Spring MVC beans or define it as the default validator for all your classes. Obviously you can define multiple `SpringValidator` instances in your Spring configuration.

The class looks like the following (shows only the relevant part):

```

public class SpringValidator implements Validator {
    ...
    public void setValidator(AnnotationValidator validator) {
        this.validator = validator;
    }

    public AnnotationValidator getValidator() {
        return validator;
    }

    public void setValidatorMap(Map<String, ValidatorParams> validatorMap) {
        this.validatorMap = validatorMap;
    }

    public Map<String, ValidatorParams> getValidatorMap() {
        return validatorMap;
    }
}

```

As you can see you should inject the AnnotationValidator you use (SpringAnnotationValidator generally) and inject a special validatorMap.

This map defines what classes this Validator supports for validation and how these classes must be validated. The ValidatorParams class can be used to define how a class must be validated. The key of the map should be the full classname of a class you support validation for. The ValidatorParams class provides several constructors and allows setter injection as well.

**Note:** The instance parameter should be ignored (this class is used by the core framework too, and instance is used there, but not needed here).

An example Spring configuration to use this altogether:

```

<bean id="SpringValidator"
class="org.javalid.external.spring.SpringAnnotationValidatorImpl">
    <constructor-arg index="0" value="org/javalid/springtest/config/jv-config.xml" />
</bean>

<bean id="myValidator" class="org.javalid.external.spring.SpringValidator">
    <property name="validatorMap">
        <map>
            <entry>
                <key><value>org.javalid.springtest.model.Employee_Lookup</value></key>
                <bean class="org.javalid.core.ValidatorParams">
                    <constructor-arg index="0"><value>1</value></constructor-arg>
                </bean>
            </entry>
        </map>
    </property>
    <property name="validator" ref="SpringValidator" />
</bean>

```

The bean named 'SpringValidator' defines the SpringAnnotationValidator using given jv-config file.

The bean named 'myValidator' defines the SpringValidator with as input a Map of classes that are supported (these classes must be annotated with JaValid annotations etc!), in this example just one class. Often you would like it to be command classes though instead of direct model objects if you require some more advanced validation.

The AnnotationValidator is injected as well. You can now use myValidator to inject in your mvc beans where you need validation for in this case the Employee\_Lookup class.

The group parameter of ValidatorParams supports one type of expression for instance: `#{validationGroups}` you can tell that the framework calls that method (getValidationGroups()) on the object in question. See the spring-web-test project that ships in the source distribution for a live example.

In some circumstances the SpringValidator is not sufficient to use or you have special validation tasks, you can use the SpringAnnotationValidator directly (inject it into your beans) and use it in the normal validation flow of Spring . You can use the SpringMessageConverter (next paragraph) to convert the messages to Spring Errors.

### 6.3 *Message converter*

To make the integration with Spring complete, the framework ships with a simple message converter **org.javalid.external.spring.SpringMessageConverter**, to convert ValidationMessage instances to the specific Spring Errors. The converter has a single method:

```
public void convertMessages(List<ValidationMessage> messages, Errors errors)
```

Just call it with the list with ValidationMessages and pass along the Spring Errors object, the ValidationMessage errors are set on the errors instance. Of course you can create a Spring bean of this class too.

That's all there is for current Spring integration!

## 7 JSF extension

The framework provides an extension for JSF (Java Server Faces), which will be discussed in this chapter.

### 7.1 Introduction

This extension allows you a flexible integration with the validation framework in a JSF web application.

In chapter 5 we have discussed how to use the Validator in a programmatic way. You are still free to use this in JSF as well, but this extension allows a different way of integrating validation – that is what this chapter is all about. Ultimately – you – decide your preferred way of doing validation, in this chapter we provide you with just another option.

JSF has its own lifecycle; a request is handled in a number of phases. For clarity, these are the phases:

- Restore View
- Apply Request Values
- Process Validations
- Update Model Values
- Invoke Application
- Render Response

At the moment of writing this document, the extension provides one integration point in these phases. Instead of integrating into the Process Validations Phase, it was decided to integrate within the Update Model Values phase (in JSF called `PhaseId.UPDATE_MODEL_VALUES`). This may look weird at first, but it is not that strange if you keep our validation framework in mind.

The power of this Validation framework is the validation of model objects. If we would integrate directly in the Process Validations Phase, our model objects (these that are in the backing beans for instance) are not up to date, because the submitted values from a user are not yet reflected in the business object. Of course it would be possible to do this, by making a copy of the original object, apply submitted values from the JSF component and then call our validation on the object. But this is a lot of unnecessary work, resulting in rather much overhead.

So instead we integrate within the Update Model Values phase in the 'after phase event'. We know that the business object is populated properly already. Most basic errors like number conversion and stuff like that are caught in the Process Validations face already, thus our object contains at least some 'sane' data here. So this is the moment we can kick off our Validation framework.

Another option for validation in JSF is also available in this framework, which has some similarities with the validation tags provided in JBoss Seam to perform validation in the Process Validations Phase. But doing it this way in our framework, will throw away a lot of the power of this framework as we only have a property to validate then, and not an entire object. But similar tags might be useful and convenient to use in certain circumstances so they will be provided nevertheless and are discussed later.

Note: This JSF extension also works with **Facelets**, which is often used in combination with JSF.

### 7.2 Integrating validation in the Update Model Values phase

In this paragraph we will describe the configuration that should be used in addition to the standard JaValid configuration.

## 7.2.1 Web application configuration

We intend to support several ways of loading / defining a validator and using it here. Since 1.1 there are two ways implemented, which are discussed below.

### 7.2.1.1 Let the JvJsfContextListener load a validator directly

To enable this type of validation, there are a few things that must be done.

1. Copy the `javvalid-jsf-dep.jar` file to your application's classpath.
2. In your `web.xml` of your web application add the following context listener.

```
<listener>
  <listener-class>org.javalid.external.jsf.JvJsfContextListener</listener-class>
</listener>
```

This listener assures you can load a validator during startup of your application, which is then used later on to perform the actual validation. In fact you can even use this listener to load a validator and use it yourself without any other integration.

Once this listener succeeds loading, an `AnnotationValidator` instance is bound in the `ServletContext` using key `JsfSupport.KEY_ANNOTATION_VALIDATOR_LOCATION`. So you can retrieve the validator by: `servletContext.getAttribute(JsfSupport.KEY_ANNOTATION_VALIDATOR_LOCATION)`. The `JsfConfiguration` instance is also stored in the `ServletContext`, but then under key: `JsfSupport.KEY_JSF_CONFIGURATION_LOCATION`.

Any errors that occur during loading are logged to a `log4j` instance and an exception is raised.

3. In your `web.xml` you *must* specify the following parameter, which tells what `javvalid-jsf` configuration file to use.

```
<context-param>
  <param-name>jv-jsf-config-file</param-name>
  <param-value>nl/test/view/jsf/jv-jsf-config.xml</param-value>
</context-param>
```

You should replace the `param-value` with the proper path to your configuration file. Do not mistake with the core configuration file, this is a separate file! This file will be discussed later.

4. Optionally you can add the following parameters in your `web.xml` file.

```
<context-param>
  <param-name>jv-config-file</param-name>
  <param-value>org/javalid/core/config/jv-config.xml</param-value>
</context-param>
<context-param>
  <param-name>jv-validator-class</param-name>
  <param-value>org.javalid.core.AnnotationValidatorImpl</param-value>
</context-param>
<context-param>
  <param-name>jv-value-to-null-if-empty</param-name>
  <param-value>>false</param-value>
</context-param>
```

`jv-config-file` is the path to your custom core configuration xml file. If you do not specify this parameter the default configuration file is used.

`jv-validator-class` is the `AnnotationValidator` you wish to use for the validation (this one is then used by the extension).

`jv-value-to-null-if-empty` can be set to `true` or `false`, to enable or disable replacing empty value "" to null (`true`) or not (`false`). See chapter 5.2 for more information.

As you can see you can fully configure the AnnotationValidators that ship with the framework through this configuration.

5. Last but not least you need to add the following PhaseListener to your faces-config.xml file.

```
<lifecycle>
  <phase-listener>org.javalid.external.jsf.JvJsfUpdateValuesPhaseListener</phase-listener>
</lifecvcle>
```

Without this listener, the previously declared stuff in web.xml would load fine, but it does not integrate with JSF then.

That's it! Your configuration is ready for integration usage now.

Obviously nothing is going to happen now, even though everything will load fine. Remember that we specified a special configuration file? It is time to discuss that now.

Important: Currently this implementation uses the message bundle defined in your faces-config.xml by default, so don't forget to specify this – it must contain the error messages from the framework. See the next paragraph to change this behaviour.

Note: Using this configuration it is not possible using the SpringAnnotationValidator with for instance the database extension that uses a Spring datasource, as the context is not set when the SpringAnnotationValidator is created manually. Instead you should use the solution discussed next paragraph.

### 7.2.1.2 Locate validator as a Spring/JSF bean

This option is identical to previously discussed option, with the exception that the validator is not created by the JvJsfContextListener, but instead it is located in Spring or JSF.

Follow the first 3 steps from the paragraph above and then follow this:

1. Set the web.xml init parameters:

```
<context-param>
  <param-name>jv-use-bean</param-name>
  <param-value>>true</param-value>
</context-param>
<context-param>
  <param-name>jv-bean-name</param-name>
  <param-value>nameBean</param-value>
</context-param>
```

This tells the listener to use a bean instead of creating one. The beanname must be defined in either Spring or JSF. If you don't use Spring it simply checks in JSF (there are no unnecessary dependencies). This configuration allows you to use a single validator defined, instead of a possible two with previous configuration.

## 7.2.2 JSF Configuration file

In the previous paragraph you configured your application for this JSF extension, for this automatic validation a special configuration file is required. Obviously the extension needs to know when to apply validation, and where.

The configuration file looks like this (example):

```

<?xml version="1.0" encoding="utf-8" ?>
<jv-jsf-validation>
  <resource-bundle name="org.javalid.core.validator.jv_messages" />
  <view id="/path.jspx">

    <validation-rule active="true"
      object="#{beanName.object}"
      groups="#{groups}"
      recurseLevel="1"
      validationPrefixPath="person"
      backingBeanName="myBeanName"
      namingContainerId="idOfParentContainer"
      ignoreUIComponentNotFound="false"

    />
    <validation-rule active="true"
      object="#{beanName2.anotherObject}"
      groups="create"
      recurseLevel="1"
      backingBeanName="sexyBeanName"
      namingContainerId="idWhatever"
      ignoreUIComponentNotFound="true"

    />
  </view>
  <view id="/pages/anotherPath.jspx">
    ...
  </view>
</jv-jsf-validation>

```

The file can contain a number of view tags. Each tag defines a path (an actual existing path in your application to your JSF page). Inside the view 1 or more validation rules can be specified, these are applied where appropriate.

### 7.2.2.1 Resource-bundle tag

With this optional tag you can tell what message bundle to use when converting ValidationMessage's to FacesMessage's. If this tag is not specified, it uses the default message bundle specified in faces-config.xml (so at least one of these two MUST be set).

### 7.2.2.2 View tag

The view tag defines a path using its id property. This path is used for matching during the Update Model Values phase, and if matched the rules are each considered for validation.

id=Should be a path to your JSF page you wish to validate automatically. Note this id must be unique for the configuration.

### 7.2.2.3 Validation-rule tag

Using this tag you can specify a validation-rule. Such a rule is either applied for a path (id) or not at all, depending on the attributes set.

Validation rules are very dynamic; each attribute can contain a literal value or a JSF expression, the latter is solved as soon as the validation rule must be applied. This allows you to decide when a rule must be applied or not, what groups must be validated and so on. The tag has the following attributes:

active=true | false, required. If true the validation rule is applied, if false it is skipped. Literal value or jsf expression.

**object**=The actual object to validate, required. Must be a JSF expression that resolves to the object that requires validation.

**groups**=String value, representing one or more groups (the groups you annotated your object with), required. You can specify multiple groups by separating them with a comma. Each group is then validated. Literal value or JSF expression.

**recurseLevel**=int value, representing the recursion depth, required. In most cases you will want to use 1 (no recursion). Literal value or jsf expression.

**validationPrefixPath**=String value, representing the path to use as a prefix during validation (see the AnnotationValidator for more info on this parameter), optional. Literal value or jsf expression.

**backingBeanName**=String value, representing the backing bean name of the object that is validated, not setting it will result in a 'guess' the component where this validation message is for, optional. Literal value or jsf expression.

**namingContainerId**=String value, representing the id of the JSF NamingContainer component where the JSF components of the object under validation reside (thus the JSF components to validate must be inside this namingcontainer), required. This is used for quickly finding the components and creating FacesMessages for the correct component. The better you define this id, the faster a component is found. Note that with simple pages you can simply use an id of a parent component.

**ignoreUIComponentNotFound**=true | false, representing what the framework must do when it cannot find a jsf component for a validationmessage. If true, adds a GLOBAL facesmessage (thus without jsf client id), if false raises an exception instead – so that you, the developer can fix this. It is recommended to use false in most cases, but during development true might be a good choice as you can see then that you are forgetting things quickly.

Now that you have defined everything properly, you can simply add declarative rules in this xml file for your validation purposes, without ever writing even one single line of validation in your backing beans.

Note: Remember that you can still use the AnnotationValidator manually in addition to this, where this might be necessary.

That's all there is for integrating this extension.

### 7.3 Direct JSF Component validation

In the previous paragraphs we have discussed the 'integrated way' of validating in JSF (or Facelets). In some circumstances you might want to use the normal way of validation JSF provides. To allow this, this extension ships with special JSF components. Make sure the javalid-jsf-dep.jar library resides on your classpath of your webcontainer.

The namespace to use is: **http://www.javalid.org/taglib/jv-ui**

The default prefix is: **jv-ui**

To include and use the component in your jsp:

```
<%@ taglib uri="http://www.javalid.org/taglib/jv-ui" prefix="jv-ui"%>
```

To include and use the component in your jsp file:

```
<jsp:root xmlns="http://www.w3.org/1999/xhtml"
xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0"
xmlns:h="http://java.sun.com/jsf/html"
xmlns:f="http://java.sun.com/jsf/core"
xmlns:jv-ui="http://www.javalid.org/taglib/jv-ui">
```

To include and use it in your facelets file:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:jv-ui="http://www.javalid.org/taglib/jv-ui">
```

No other configuration is needed for usage.

### 7.3.1 validateAll tag

The validateAll tag contains two properties, both are required. These are:

`active=true | false`, if true the validation is applied.

`group=Group` to validate, String value.

Both support JSF expressions.

All components inside this tag are scanned, to see if they implement the JSF interface `EditableValueHolder`. If they do, a `JvComponentValidator` is added which will perform the actual validation for such component (note: added, other validators can still be used on the same component in addition). Whether validation is really performed depends on whether the tag is active or not.

In the example below, validation is applied to both `firstName` and `lastName` components (these are `EditableValueHolder` components) for the group `create`.

```
<jv-ui:validateAll active="true" group="create">
  <h:outputText value="First name" />
  <h:inputText value="#{PersonOverviewBean.person.firstName}" id="firstName" />
  <h:message for="firstName" />

  <h:outputText value="Last name" />
  <h:inputText value="#{PersonOverviewBean.person.lastName}" id="lastName"
required="true" />
  <h:message for="lastName" />
</jv-ui:validateAll>
```

In certain cases `validateAll` may not work properly (for example with lists e.g. with `ui:repeat` from facelets generating input components). In that case you should use the simple `jv-ui:validate` tag instead.

### 7.3.2 validate tag

This is a validation tag and thus must be used as a validator for input components. Using this tag you can validate a single property. The difference with `validateAll` / `validate` is that the latter is executed in the `processValidations` phase, while the first is executed in the `update values` phase.

```
<ui:repeat value="#{EmpBean.emp.children}" var="child" >
  <tr:outputText value="Name" />
  <tr:inputText value="#{child.name}" required="true" simple="true" id="sexyId">
    <jv-ui:validate active="true" group="edit02"/>
  </tr:inputText>
  <tr:message for="sexyId" />
</ui:repeat>
```

The example above (facelets example) generates input components for a list of children and assures each component is validated.

The next paragraph points out a couple of handy methods that ship with the extension, which you can easily use yourself. Who wants to write boiler code if someone else did this already? ☺

## 7.4 Some handy methods

This paragraph just adds a short description of handy (jsf related) methods you might be interested in for using in your own project.

### 7.4.1 org.javalid.external.jsf.JsfSupport

This class provides several JSF related methods.

Note that the ones described in this document are likely to remain in later releases – others not discussed here might not (or change). However use at your own risk, you could also copy the code from them to your own project.

#### **public ResourceBundle getResourceBundle(Locale locale, String bundleName)**

Can be used to retrieve a ResourceBundle in given Locale, bundleName should be a path to a properties file. Raises JavalidException if bundle was not found.

#### **public boolean isJsfExpression(String value)**

Returns true if given value looks like a JSF expression, no more than that, it is not resolved. Null String value returns false too. You are recommended to use JsfSupport.newJsfFacade() instead, which will return a facade to use for whichever JSF version is used. If you are already on JsfSupport you can use getJsfFacade() directly as well. You can then use the isExpression method on the facade. Since 1.0.1.

#### **public String stripJsfTagsFromExpression(String expression)**

Strips a jsf expression of its tags and returns that value. If expression is null, returns null. You are recommended to use JsfSupport.newJsfFacade() instead, which will return a facade to use for whichever JSF version is used. If you are already on JsfSupport you can use getJsfFacade() directly as well. You can then use the removeExpression method on the facade. Since 1.0.1.

#### **public FacesMessage createFacesMessage(ResourceBundle bundle, String jvMessage, Object[] paramValues, Locale locale)**

Creates a FacesMessage for given 'jvMessage' (this can be any code from a resource), parameter values and in given locale. The bundle is used to lookup the message.

### 7.4.2 org.javalid.external.jsf.JsfMessageConverter

This class is especially useful in the case where you perform programmatic validation, using one of the public methods in this class you can convert ValidationMessage's automatically to FacesMessages and these are added to the FacesContext.

Please consult the API documentation for details.

## 8 Database extension

This extension allows you to configure validation constraints against a database of your choice. This extension is new since JaValid 1.1.

### 8.1 Configuration and usage

To use this extension, you need to do the following:

1. First you need to add the javalid-db-dep.jar library in your application's classpath.
2. The second step is to enable the database extension:

```
<extension name="database"
    extension-loader-
class="org.javalid.external.db.extension.JavalidExtensionDatabaseImpl">
  <parameters>
    <parameter name="configFile"
      value="org/javalid/external/db/config/javalid-db-example.xml"/>
  </parameters>
</extension>
```

As you can see the extension is named 'database' , you must not change this name. You also need to add a configuration parameter named 'configFile' which points to your database configuration xml file (which is discussed below).

3. Create a database configuration file, this is an xml file which defines the datasource / connection to use. It also allows you to define queries.

This is a sample xml file, you must change it to reflect your own situation:

```

<?xml version="1.0" encoding="utf-8" ?>
<javalid-db-ext>

  <!-- type can be jndi or spring -->
  <datasources>
    <!-- <datasource id="myDB" type="jndi" name="a" /> -->
    <!-- Does create an internal DS based on apache commons pooling using
connection data -->
    <connection id="myConn" user-name="sa" password=""
url="jdbc:hsqldb:mem:testdb" driver="org.hsqldb.jdbcDriver" />
  </datasources>

  <sql-queries>
    <sql-query id="myQuery" ref-ds="myConn">
      select count(1) from my_table where id=${this};
    </sql-query>

    <sql-query id="anotherQuery" ref-ds="myConn">
      <![CDATA[
        select p.val
        from product p
        where p.id=${this} and p.value in (select x from another_table)
      ]]>
    </sql-query>

    <sql-query id="sexyQuery" ref-ds="myConn" static-
field="org.javalid.test.db.unit.TestDbConfiguration.FIELD_NAME" />
  </sql-queries>

```

The first thing you need to decide is what type of datasource you will use. You can use a real datasource provided by your application server for instance, or a manually created datasource. We will discuss the tags below.

### 8.1.1 Datasource tag

A datasource can be defined using the `<datasource>` tag (inside the `<datasources>` tag). Currently two types are supported: Datasource to be located by JNDI (Java Naming Directory Interface) or by Spring (Spring extension must be enabled).

**id**=The id of the datasource, this must be a unique identifier. This will be used to refer to this datasource from within your validation constraints.

**type**=Type of datasource, values: `jndi` or `spring`.

**name**=Name of the datasource or Spring bean that represents the datasource. For instance `jdbc/myApplicationDS` for `jndi`, and `SpringDataSource` for `Spring`.

For JNDI you might want to specify its initialization parameters (though generally you don't need to as `jndi.properties` is on the application server's classpath most times already). You can do this by either providing a `jndi.properties` file on the classpath yourself or adding properties using the `<jndi-properties>` subelement of `datasource`. For instance:

```

<datasource>
  <jndi-properties>
    <jndi-property name="java.naming.provider.url" value="ormi://localhost" />
    <jndi-property name="java.naming.factory.initial"
value="com.evermind.server.ApplicationClientInitialContextFactory" />
  </jndi-properties>
</datasource>

```

For exact parameters and their values please consult the JNDI documentation found on <http://java.sun.com> as well as the documentation of your application server. Again generally you do NOT need to specify them.

### 8.1.2 Connection tag

While the `datasource` tag is used to point to real datasources of your application (server), the `<connection>` tag allows you to create a datasource by hand. JaValid will create this datasource using the Apache Commons Pooling library and use it to perform the validation constraints.

Note: You could use this technique for testing purposes too if you run your tests outside the application server for instance.

An example connection definition might look like this (this one is for the in-memory Hsqldb database):

```
<connection id="myConn" user-name="sa" password="" url="jdbc:hsqldb:mem:testdb"
driver="org.hsqldb.jdbcDriver" />
```

The following attributes are defined for the connection tag:

id=The id of the connection (same as id of datasource element), this must be a unique identifier. This will be used to refer to this datasource from within your validation constraints.

user-name=The user name for the database.

password=The password to use.

url=JDBC URL to use for your database (consult your database's jdbc drivers documentation).

driver=The driver class to use

### 8.1.3 Sql-query tag

This tag can be used to define an SQL query. What queries can be defined exactly depends on the database annotation used. The query tag has the following attributes:

id=Unique name of the query for the datasource it is meant for

ref-ds=Reference to the id of a datasource or connection.

static-field=If set you can refer to a static field in a class for the query. If this attribute is set you cannot define the query inside the body of the tag anymore.

Note that this tag is optional, you can also define the queries directly with the database annotations. Which way you use is entirely up to your own preference. The database extension supports both xml defined or annotation defined queries (or both).

The following example shows two query definitions, one as a query the second as a reference to a static String field which contains the query.

```

<sql-queries>

  <sql-query id="anotherQuery" ref-ds="myConn">
    <![CDATA[
      select p.val
      from product p
      where p.id=${this} and p.value in (select x from another_table)
    ]]>
  </sql-query>

  <sql-query id="sexyQuery" ref-ds="myConn" static-
field="org.javavid.external.test.db.TestDbConfiguration.FIELD_NAME" />
</sql-queries>

```

The first query is defined with id 'anotherQuery' for a datasource that has an id of 'myConn'. The query itself is set between xml's <![CDATA[ ]]>. Annotations can refer to this query using its id.

The second query with id 'sexyQuery' for a datasource with id 'myConn' is defined as a static query. It tells it must use the field named FIELD\_NAME in the class 'org.javavid.external.test.db.TestDbConfiguration', which contains the query. Again annotations can refer to this query using its id.

## 8.2 Validation annotations

In the previous paragraph we discussed the configuration of the database extension. That is all beautiful, but without an annotation to use it, quite useless.

Currently the distribution ships with a single annotation, this might sound as 'nothing' but it is not if you know how to use the annotation well.

In the future more annotations may be added depending on people's (you!!) need, so let yourself be heard!

### 8.2.1 @DbNumCheck

This annotation can be specified on both methods (get) or fields and checks the value of such method / field against the database. The value returned from the query must be a whole number (it will be read in as a java.lang.Long). The annotation has the following attributes:

**refDsId**=The id of a datasource / connection defined in the xml configuration (required).

**refQueryId**=The id of a query defined for the datasource you specified in refDsId. You can either use this property or set the query attribute, one of them is required. Defaults to an empty String.

**query**=Actual sql query to use. Defaults to "", either this one or refQueryId is required.

**value**=The value the query must return (java.lang.Long). Your query must return a numeric result (generally you would use a count(\*) type of query to determine if something is valid or not; the value returned there must adhere to the mode() property). For instance if the mode() property is set to MODE\_EQUALS, it must match this property.

Note that this one is required for all MODE\_ constants, except MODE\_BETWEEN, you must specify minValue() and maxValue() then instead. The value defaults to 1 (1L).

**minValue**=Must be set when mode() is set to MODE\_BETWEEN, this represents the minimum value then (inclusive). Defaults to 1 (1L).

**maxValue**=Must be set when mode() is set to MODE\_BETWEEN, this represents the maximum value then (inclusive). Defaults to 1 (1L).

**mode**=The mode of the check, defaults to DbNumCheck.MODE\_EQUALS. Use one of the following:

- DbNumCheck.MODE\_EQUALS (value of method/field must equal value() property)
- DbNumCheck.MODE\_NOT\_EQUALS (value of method/field must NOT equal value() property)
- DbNumCheck.MODE\_LESS\_THAN (value of method/field must be LESS than value() property).
- DbNumCheck.MODE\_MORE\_THAN (value of method/field must be MORE than value() property).
- DbNumCheck.MODE\_BETWEEN (value of method/field must be between minValue() and maxValue() properties, both inclusive).

**customCode**=Defaults to empty string. If you need a specialized message for the validation error of this annotation, set it to a custom property (from one of your own property files).

**applyToGroups**= See definition in 4.2.2.

**globalMessage**=See definition in 4.2.2.

So much for the theory let's look at a few examples:

```
@JvGroup (groups={"simpleProduct"})
@NotNull
@DbNumCheck (refDsId="myConn",query="select count(*) from product_category where
id=${this.id}")
public ProductCategory getProductCategory() {
    return productCategory;
}
```

This check validates that the id of the ProductCategory that is annotated does exist in the product\_category table (after all the query would return 1 if it can find it). The default mode is equals, and the default of the value() property is 1. Note the expression in the query, expressions are discussed in the last paragraph of this chapter.

```
@JvGroup (groups={"idExample"})
@DbNumCheck (refDsId="myConn",refQueryId="anotherQuery")
public ProductCategory getProductCategory07() {
    return productCategory07;
}
```

This check does the same validation as above, except it refers to a query that is defined in the xml configuration file.

```
@JvGroup (groups={"betweenProduct"})
@DbNumCheck
(refDsId="myConn",mode=DbNumCheck.MODE_BETWEEN,minValue=1,maxValue=2,query="select
count(*) from product_category where id > ${parent.someWorkingValue}")
public ProductCategory getProductCategory03() {
    return productCategory03;
}
```

This last example performs a check that the value returned from the query is between 1 and 2, this time the query is written out again but could also be put into the configuration file.

## 8.3 Expressions

The database extension allows you to use expressions inside your queries. This provides you with a very flexible and powerful way of writing your queries as it allows you to traverse the objects that are annotated without any limitations.

The expression language used is the same as the one used in JSP (Java Server Pages) or JSF (Java Server Faces) expressions, but it is not coupled to any of these techniques.

Instead JaValid uses the powerful expression library JUEL (Java Unified Expression Language) hosted on sourceforge (<http://juel.sourceforge.net>), written by Christoph Beck.

Inside your query you can use two special keywords to get access to the value you annotated or its declaring class. These are:

- this
- parent

The first keyword represents the field or method's value that is annotated by a database annotation. The parent keyword represents the class that declared the method / field you annotated. By using these keywords you have access to your object graph.

For instance if you annotated something like:

```
@ValidateDefinition
public class Example {
    @DbNumCheck( ... )
    public Order getOrder() { .. }
}
```

If you would use the this query inside your query it would refer to the Order instance. If you would use the parent keyword it would refer to the Example instance. You could easily write powerful queries now. For example:

```
select count(*)
from example
where order_id=${this.id}
and category=${parent.anotherInstance.name}
```

Remember that all unified expression types are supported, so you can access lists or arrays using (0) for the first element for instance, or maps using ['key'] and so on.

## 9 Add custom extension

In the previous chapter you've learned that JaValid uses extensions on top of its core. In this chapter you will learn how to add a custom extension yourself. Generally you would add an extension if you want to add specialized custom annotations that cannot be handled by any of the available extensions or core.

An example could be for instance: You need an annotation to contact a webservice to perform a validation. To do so you would need a lot of extra information not available in JaValid itself, thus you would add a custom extension which does have that information. Creating annotations for this extension would be no different than adding any other annotation, except your validation handler, which would know how to get this extra information.

You need to do the following:

1. A class which implements the interface: `org.javalid.core.extension.JavalidExtension`

The interface looks like this (stripped off comments):

```
package org.javalid.core.extension;

import java.util.Map;
import org.javalid.core.validator.JvConfigurationWrapper;

public interface JavalidExtension {

    public void init(Map<String,String> parameters, JvConfigurationWrapper wrapper);

    public void configurationLoadingComplete(JvConfigurationWrapper wrapper);

    public void beforeValidation(JvConfigurationWrapper wrapper);

    public void afterValidation(JvConfigurationWrapper wrapper);

    public void destroy();
}
```

The interface defines 4 methods:

- `init(..)`, which is called by JaValid once the configuration is loaded. This is the moment where you initialize your extension (such as loading configuration etc). The map contains parameters you can specify in JaValid's own configuration.
  - `Destroy()` is called once the validators own `destroy()` method is called, after that the `AnnotationValidator` is not usable anymore. It allows extensions to cleanup if needed.
  - The `beforeValidation(..)` method is called by the framework *before* validation is performed. So this would be each time a user would call the `validator.validateObject(..)` object. You could initialize certain resource types that are needed by your extension for example.
  - The `afterValidation(..)` method is called by the framework *after* validation is done. Generally you should use this for cleanup of previously set resources.
  - The method `configurationLoadingComplete(..)` is called once an `AnnotationValidator` is fully loaded (generally at the end of the constructor), but this can be changed if needed. This is useful where for instance you require a Spring context be available for your initialization of this extension.
2. Register your extension, in the `<extensions></extensions>` part of the main configuration of JaValid and give it a unique name. It's here that you can pass parameters to your configuration (which end up in the `Map<String,String>` of the `init()` method previously discussed). For examples see the extensions of JaValid itself.

Finally you can find your extension and your custom annotations by using one of the methods exposed on the `org.javalid.core.validator.JvConfigurationWrapper` class. This class is passed to your custom annotation's validators. Use:

- `public boolean isExtensionLoaded(String nameExtension)`

- `public JavalidExtension getExtension(String nameExtension)`

The name is the name you gave it in the configuration of the `<extension></extension>` tag. You can then safely cast it to your implementing class and do any magic you wish to do there!

As you can see, its really not hard to add a custom extension! This allows you to add powerful annotations for anything you can think of.

See the database extension for a good example, how it can be done.

Please let us know if you add a nice extension yourself, it might be well worth it to add it to the distribution and share it with everyone!

## 10 Examples

Examples are maintained on the website <http://www.javalid.org> and can also be found in the unit tests that ship in both the source/binary distribution.

# 11 Release notes

-----  
Changes from 1.1rc1 to 1.1  
-----

Bugs fixed:

- Loading of Spring defined datasource fails
- toString() ValidationMessage causes a nullpointer if array with null values is passed.
- If JSF is detected, incorrect elcontext is loaded for database extension

FEATURES/ENHANCEMENTS:

- Loading of config file in debug mode does not print stacktrace anymore, as this could be confusing (just prints a message instead).

KNOWN ISSUES:

- LinkageError with javax.el.\*  
If you deploy to an application server that already has an el-api.jar in its classpath (such as Tomcat 6 or JBoss 4.2 etc), you may encounter a LinkageError classloader issue. To solve this you must NOT deploy an el-api yourself with your application, nor deploy the juel-api.jar file (juel-impl is required!).  
Another solution might be to force classloading your (web)application libraries first. Check the documentation of your container on how to do this.

-----  
Changes from 1.0.1 to 1.1-rc1  
-----

BUGS fixed:

- Fixed webtest of spring, was broken in Tomcat 5.5
- Fixed bug with jv-ui:validate tag, which only resolved an expression once.
- 1967984 Fixed incorrect documentation for jv-ui:validate component

FEATURES/ENHANCEMENTS:

- Introduced the option to add your own extensions to JaValid (don't confuse with custom annotations). Allowing you to add anything to it you like. Extensions from 1.0 (Spring / JSF) are refactored to the new extensions standard as well. Check the docs on how to add a new extension!
- Added Database integration validation for datasources (from container or handmade)
- Added @DbNumCheck annotation which provides a powerful database check, including specialized EL support in the queries.
- Added JavalidValidationHandler to be passed when calling a validateObject() method. Allows for customization before / after validation.
- Added plural annotations for all existing validation annotations, that is you can use a validation annotation multiple times on the same field/method e.g @MinLengths(values={@MinLength(..),@MinLength(..)}). The original way from 1.0 works still fine too. Check the docs how to add this to your own annotations!  
(SF: 1992025 Support use of similar annotations on same field/method)
- Added @DateCheck annotation, which allows to check dates (equals, more, less etc).
- Added support for field validation through the jv-ui tag (was only method)
- ValidatorSupport, ReflectionSupport, XMLSupport, Jsfsupport, JsfsupportMessageConverter refactored, all methods are static final and the constructors are private.
- Added a new web test project which has jsf 1.2 with facelets, as sample but

as a test as well.

KNOWN ISSUES:

- LinkageError with javax.el.\*

If you deploy to an application server that already has an el-api.jar in its classpath (such as Tomcat 6 or JBoss 4.2 etc), you may encounter a LinkageError classloader issue. To solve this you must NOT deploy an el-api yourself with your application, nor deploy the juel-api.jar file (juel-impl is required!).

Another solution might be to force classloading your (web)application libraries first. Check the documentation of your container on how to do this.

-----  
Changes from 1.0 to 1.0.1  
-----

BUGS fixed:

None.

FEATURES/ENHANCEMENTS:

1963863 Code is now independent of JSF version by introducing Jsffacade

-----  
Changes from 1.0-rc2 to 1.0  
-----

BUGS fixed:

None.

FEATURES/ENHANCEMENTS:

1954319 Provide selective global message support

1955130 Add BigInteger / BigDecimal support

1956675 Add Collection size check annotation

-- Refactored JvValidator to JavalidValidator (check custom annotations!)

-- Added more examples

NOTES:

None.

-----  
Changes from 1.0-rc1 to 1.0-rc2  
-----

BUGS fixed:

1944076 Inheritance validation fails in some cases

1946399 Cannot use core library directly, noClassDefFoundError

FEATURES/ENHANCEMENTS:

-- Added a few more examples.

NOTES:

None.

## 12 Upgrading JaValid

This chapter describes what you need to do when you start using a newer version of JaValid compared to a previous one.

### 12.1 Upgrade 1.0 to 1.1

- Replace your current configuration file with the new 1.1 version one. See 3.1 for the new configuration file.
- Enable the extensions you had previously enabled (Spring or Jsf) by removing the comment
- If you added any custom annotations yourself, add these again to this file – but make sure you add the extra attribute: `supports-plural="false"` (as yours do not yet support plural versions of themselves). See 4.2.1 and 4.3 to learn how to make your own annotations plural too (its easy).
- Optionally, if you provided different implementation classes for Jsf or Spring you may need to change the parameter values in the new configuration file.

## 13 Thanks

I would like to thank everyone that helped me improving JaValid in this chapter.

I'd like to especially thank Christoph Beck (author of JUEL and co-author of STAN4J) who helped me on using JUEL and how to put it to best use.

I also like to thank Maarten Buijter for reviewing this document and get the worst grammar errors out.

I also like to thank some of my colleagues who helped on several ideas as well as to improve parts of the implementation.

Finally I'd like to thank you, for taking the time to read the documentation, and for using JaValid!

If you have any comments, feedback or whatsoever please contact me at: [feedback@javalid.org](mailto:feedback@javalid.org) or drop a comment on the weblog of JaValid's website: [www.javalid.org/wps/](http://www.javalid.org/wps/)