# JValidate, a validation library

Version:

# Table of Contents

# Chapter 1. Introducing JValidate

## 1.1. What is JValidate?

JValidate is a library for validating javabeans. Java annotations or XDoclet-tags are used define validation rules for properties of javabeans. A validation engine validates the javabeans. Furthermore it provides utility classes to propagate and transform validation messages to the user interface.Several, web-framework specific, components are included to present the validation messages.

## 1.2. Why do we need JValidate

JValidate solves a problem that many (web) developers faces. We want to define our validation once and only once, and by preference not in the UI. Further, we do not want to depend on whatever validation mechanism the web-framework is using.

## 1.3. What are the goals of JValidate

With JValidate we tried to create a generic validation engine, which requires no coupling to validation code in the javabeans itself. Other design goals of JValidate are

- simplicity
- consistency
- ease of use

We strived for simplicity and ease of use for the developer in defining the validation rules. Consistency in defining these rules once and only once and consistency in the mechanism used to validate and presentation of validation messages.

Try JValidate yourself to see if we succeeded in accomplishing these goals!

# Chapter 2. Getting started with JValidate

In this chapter we show you step by step, by example, how you can use Jvalidate to validate your java beans and how to present validation message's in the UI.

## 2.1. Download and install

First download and install the required libraries. You can download the JValidate jars from the sourceforge site. Which libraries you need is depending of which JDK and which web-framework, if any, you are using. JValidate consists of the following libraries.

**Table 2.1. JValidate libraries**

| | |
|---|---|
| JValidate-{version}.jar | Contains the core validation classes. |
| JValidate-annotations-{version}.jar | Contains jvalidate annotations, only for JDK 1.5 and higher. |
| JValidate-doclet-{version}.jar | Contains xdoclet library and ant-tasks for JDK 1.4 and higher. Only required at buildtime. |
| JValidate-jsf-{version}.jar | Contains jsf components for showing validation messages. |

JValidate is using a set of third party libraries.

**Table 2.2. third party libraries**

| | |
|---|---|
| log4j | 1.1.3 |
| commons-beanutils | 1.7.0 |
| commons-lang | 2.1 |
| commons-collections | 2.1 |
| commons-logging | 1.0 |
| gnu-regexp | 1.1.4 |
| xdoclet | 1.2.3 (Only buildtime required when using jvalidate-doclet tags) |

## 2.2. Define the validation rules

The second step is defining your validation rules. If you are using JDK 1.5, you can use java annotations, otherwise you have to use xdoclet tags.

Section 2.1 describes how to use annotations, section 2.2 describes how to use XDoclet tags. For both section's we define a customer and an address javabean, for which we define the validation rules.

### 2.2.1. Annotation with java 1.5 annotations

In the source code of customer we add annotions for validation, first we start annotating the class, with the @ValidateClass tag, to mark it as a validated class.

```
...

  @ValidateClass
  public class Customer {

  ...

  }
```

Now we start adding validation rules for the properties, we do this by annotation the getters.

The firstname of the customer must have a value and may not be blank, we define this validation with the @ValidateIsNotBlank annotation.

```
...

  @ValidateIsNotBlank
  public String getFirstName() {
        return firstName;
  }
...
```

Furthermore, firstname may not be longer then 20 characters, we define this validation with the @ValidateMaxLength annotation and a annotation attribute length.

```
...

  @ValidateIsNotBlank
  @ValidateMaxLength(length=20)
  public String getFirstName() {
        return firstName;
  }

...
```

For each validation it's possible to add a user defined error code, this is done with the annotation attribute error-Code. This is an optional attribute, when not defined the default errorcodes are used. See Appendix A.

```
  @ValidateIsNotBlank(errorCode="firstname.isrequired")
  @ValidateMaxLength(length=20, errorCode="firstname.tolong")
  public String getFirstName() {
        return firstName;
  }
```

The address class has a (dutch) zipcode which must conform a specified pattern

```
...
```

```
@ValidateClass
public class Address{

   ...

 @ValidatePattern(pattern="^[1-9]\\d{3}[- ]?[a-zA-Z]{2}$", errorCode="invalid.zipcode")
 public String getZipCode() {
     return zipCode;
 }

 ...

}
```

Address as a member of customer must also be validated. We annotate the getAddress() method with the @ValidateIsValid annotation to validate the address. Its also possible to use the @ValidateIsValid for collections or arrays of user defined, validated, classes.

```
...

  @ValidateIsValid
  @ValidateIsRequired
  public Address getAddress() {
        return address;
  }
...
```

We are done with adding validation rules to our customer, next step is JUnit testing our validations. If you are using java annotations you can skip the XDoclet section.

## 2.2.2. XDoclet

In this section we describe how XDoclet tags are used to define the validations. When using XDoclet two steps are required, first define validations with xdoclet tags, the second step is generating ClassValidators with XDoclet Ant integration.

### 2.2.2.1. XDoclet tags

In the source code of customer we add XDoclet tags for validation, first we start tag the class, with the @validate.class tag, to mark it as a validated class.

```
...

/**
 * @validate.class
 *
 */
public class Customer{

...

}
```

Now we start adding validation rules for the properties, we do this by adding XDoclet tags to the getters.

The firstname of the customer must have a value and may not be blank, we define this validation with the @validate.isnotblank tag.

```
...

 /**
   * @validate.isnotblank
   *
   * @param firstName The firstName to set.
   * @return Returns the firstName.
  */
   public String getFirstName() {
     return firstName;
   }

...
```

Furthermore, firstname may not be longer then 20 characters, we define this validation with the @validate.length XDoclet tag and a attribute value.

```
...

  /**
   * @validate.isnotblank
   * @validate.maxlength value="20"
   *
   * @param firstName The firstName to set.
   * @return Returns the firstName.
  */
   public String getFirstName() {
     return firstName;
   }

...
```

For each validation it's possible to add a user defined error code, this is done with the annotation attribute errorCode. This is an optional attribute, when not defined the default errorcodes are used. See Appendix A.

```


...

  /**
   * @validate.isnotblank errorcode="firstname.isrequired"
   * @validate.maxlength value="20" errorcode="firstname.tolong"
   *
   * @return Returns the firstName.
   */
   public String getFirstName() {
     return firstName;
   }

...
```

The address class has a (dutch) zipcode which must conform a specified pattern

```


...
```

```
/**
 * @validate.class
 *
 */
public class Address{

...

  /**
   * @validate.pattern value="^[1-9]\\d{3}[- ]?[a-zA-Z]{2}$" errorcode="zipcode.invalidformat"
   *
   * @return Returns the zipcode.
   */
    public String getZipCode() {
      return zipCode;
   }

...

  }
```

Address as a member of customer must also been validated. We do this by tagging the getAddress() method with the @validate.isvalid annotation. Its also possible to use the @validate.isvalid for collections or arrays of user defined, validated, classes.

```
...
public class Customer{

   ...

  /**
   *
   * @validate.isvalid
   * @validate.isrequired
   *
   * @return Returns the Address.
   */
   public Address getAddress() {
        return address;
  }
...
```

We are done with adding validation rules to our customer, next step is generate <ClassName>ClassValidators.java files with XDoclet.

#### 2.2.2.2. Generate ClassValidators with XDoclet

The second step is generating ClassValidators with XDoclet. For this step the jvalidate-xdoclet.jar is required.

```
...



<taskdef name="validates" classname="xdoclet.modules.knowlogy.validation.ValidationDocletTask" >
    <classpath>
       <path refid="validate.classpath"/>
    </classpath>
</taskdef>
```

```
<target name="create.validation">
  <mkdir dir="${basedir}/target/java/generated" />
  <validates destdir="${basedir}/target/java/generated" force="true" verbose="true">
      <fileset dir="${basedir}/src/test/java" includes="**/*.java"/>
      <validate/>
  </validates>
</target>
```

The validates taskdef creates <ClassName>ClassValidator.java files, which must be compiled and packaged with your application.

## 2.3. JUnit testing validations

This section explains how validations can be junit tested. Its also introduces us with the ValidationEngine and Messages class and illustrates their use. So lets continue.

Now were done with adding the validation rules we create a JUnit test to check our validations.

```
public class CustomerValidationTest extends TestCase {


    public void testValidateFirstName(){

        Customer customer = new Customer();

        Messages messages = new MessagesImpl();
        ValidationEngine.validate(customer, messages);

        Message firstNameMessage = messages.getMessage(customer, "firstName");

        assertNotNull("expected a first name validtion message ",firstNameMessage);
        assertEquals("expected a firstname.isrequired errorcode","firstname.isrequired", firstNameMess

        customer.setFirstName("a-name-which-is-longer-then-20-chars");

        messages.clear();
        ValidationEngine.validate(customer, messages);
        firstNameMessage = messages.getMessage(customer, "firstName");

        assertNotNull("expected a first name validation message", firstNameMessage);
        assertEquals("expected a firstname.tolong errorcode","firstname.tolong", firstNameMessage.getM

        customer.setFirstName("Adrian");

        messages.clear();
        ValidationEngine.validate(customer, messages);
        firstNameMessage = messages.getMessage(customer, "firstName");
        assertNull("expected no validation message for firstname",firstNameMessage);


    }
}
```

Our JUnit test shows how to test the validation of the firstname of the customer. We create an instance of Customer that we gone validate. With ValidationEngine.validate the customer object is validated, along with the customer instance goes a Messages instance. The ValidationEngine validates the customer object and add validation messages, if any, to the messages object.

We can retrieve the validation message for a specific object and a specific property with the getMessage method. In the test we retrieve the message for the firstname property.

## 2.4. Presenting the validation messages

In this section we describe how we to propagate the messages back to the front-end. There are of course several types of front-ends, for a number of specif front-end frameworks we provide utility classes and components to integrate with those frameworks.

### 2.4.1. Java Server Faces

```
class CustomerService{


    public void saveCustomer(Customer customer)throws ValidateException{

       Validators.validate(customer);

       customerDao.saveCustomer(customer);
    }

}

class CustomerController{



   public String saveAction(){

       String outcome = "ok";
       try{
         customerService.saveCustomer(customer);
       }catch(ValidationException ve){
         FacesContext ctx = FacesContext.getCurrentInstance();
         MessagesUtil.convert(ctx, ve.getMessages());
         outcome = "";
       }
       return outcome;
   }

}
```

# Chapter 3. Advanced validations

## 3.1. Using a custom validator

## 3.2. Grouped validation

## 3.3. Adding programmatic validations

Its possible to add validations to the defined validations. There are two possible strategies. Which one you choose depends on what you want to validate.

### 3.3.1. Implement the Validatable interface

One possible strategy is to implement the validatable interface. For this interface you must implement the `validate(Messages messages)` method, in which a custom validation can created. If a class is implementing this interface, it is called by the `Validator`

This strategy can be used when validation requires internal data, which is not available at the outside of the class.

A possible disadvantage is coupling your javabean to the validating library.

## 3.4. User defined annotations

## 3.5. Integrating with Hibernate

## 3.6. Using the metadata

# Chapter 4. Under the hood

# Appendix A. Annotations

This appendix lists in alphabetic order all the annotions which are default available.

**Table A.1. ValidateClass**

| | |
|---|---|
| Name | @ValidateClass |
| Description | Marker to indicate that its possible to validate this class |

**Table A.2. ValidateAllowedValues**

| | |
|---|---|
| Name | @ValidateAllowedValues |
| Description | Validates if a String is one of the allowed values |
| Types | String |
| **Properties** | |
| allowedValues | required, |
| errorCode | Optional, default value = "@todo" |

**Table A.3. ValidateCustom**

| | |
|---|---|
| Name | @ValidateCustom |
| Description | Validates a property with a custom validator |
| Types | String |
| **Properties** | |
| validatorClassName | required, fully qualified classname of the validator which must implement the CustomValidator interface |
| errorCode | Optional |

**Table A.4. ValidateIsNotBlank**

| | |
|---|---|
| Name | @ValidateIsNotBlank |
| Description | Validates if a String property is not null or an empty string |
| Types | String |
| **Properties** | |
| errorCode | Optional |

**Table A.5. ValidateIsRequired**

| Name | @ValidateIsRequired |
|---|---|
| Description | Validates if a property is not null |
| Types | All types |
| **Properties** | |
| errorCode | Optional |

**Table A.6. ValidateIsValid**

| Name | @ValidateIsValid |
|---|---|
| Description | Validates if a property of a user defined type or collection/array of userdefined types is valid using the validator for that type |
| Types | User defined or collection/array with user defined types |
| **Properties** | |
| errorCode | Optional |

**Table A.7. ValidateMaxLength**

| Name | @ValidateMaxLength |
|---|---|
| Description | ValidateMaxLength |
| Types | String,Integer,Long, int, long |
| **Properties** | |
| length | required, maximum length of the string (or string represenation of the number) |
| errorCode | Optional |

**Table A.8. ValidateMaxSize**

| Name | @ValidateMaxSize |
|---|---|
| Description | Validates if a number is not greater than size |
| Types | Integer, Long, int, long |
| **Properties** | |
| size | required, the maximum size including |
| errorCode | Optional |

**Table A.9. ValidateMinLength**

| Name | @ValidateMinLength |
|---|---|
| Description | Validates if a String property is not null or an empty string |
| Types | String |
| **Properties** | |
| length | Required, the minimum length of a string |
| errorCode | Optional |

**Table A.10. ValidateMinSize**

| Name | @ValidateMinSize |
|---|---|
| Description | Validates if a number is not less then size |
| Types | Integer, Long, int, long |
| **Properties** | |
| size | Required, the minimum size (including) |
| errorCode | Optional |

**Table A.11. ValidatePattern**

| Name | @ValidatePattern |
|---|---|
| Description | Validates if a String conforms the pattern |
| Types | String |
| **Properties** | |
| pattern | Required, pattern (requlr expression) to which the string must match |
| errorCode | Optional |

# Appendix B. XDoclet tags

**Table B.1. validate.class**

| | |
|---|---|
| Name | @validate.class |
| Description | Marker to indicate that its possible to validate this class |

**Table B.2. validate.allowedvalues**

| | |
|---|---|
| Name | @validate.allowedvalues |
| Description | Validates if a String is one of the allowed values |
| Types | String |
| **Properties** | |
| value | required, string values seperated by , |
| errorcode | Optional, default value = "@todo" |

**Table B.3. validate.custom**

| | |
|---|---|
| Name | @validate.custom |
| Description | Validates a property with a custom validator |
| Types | String |
| **Properties** | |
| validatorclassname | required, fully qualified classname of the validator which must implement the CustomValidator interface |
| errorcode | Optional |

**Table B.4. validate.isnotblank**

| | |
|---|---|
| Name | @ValidateIsNotBlank |
| Description | Validates if a String property is not null or an empty string |
| Types | String |
| **Properties** | |
| errorcode | Optional |

**Table B.5. validate.isrequired**

| Name | @validate.isrequired |
|---|---|
| Description | Validates if a property is not null |
| Types | All types |
| **Properties** | |
| errorcode | Optional |

**Table B.6. validate.isvalid**

| Name | @validate.isvalid |
|---|---|
| Description | Validates if a property of a user defined type or collection/array of userdefined types is valid using the validator for that type |
| Types | User defined or collection/array with user defined types |
| **Properties** | |
| errorcode | Optional |

**Table B.7. validate.maxlength**

| Name | @validate.maxlength |
|---|---|
| Description | Validate if the string does not exceed the maximum length |
| Types | String |
| **Properties** | |
| value | required, maximum length of the string |
| errorcode | Optional |

**Table B.8. validate.maxsize**

| Name | @validate.maxsize |
|---|---|
| Description | Validates if number <= maxsize value |
| Types | Integer, Long, int, long |
| **Properties** | |
| value | required, the maximum size |
| errorcode | Optional |

**Table B.9. validate.minlength**

| Name | @validate.minlength |
|---|---|
| Description | Validates if a String.length >= minlength value |
| Types | String |
| **Properties** | |
| value | Required, the minimum length of a string |
| errorcode | Optional |

**Table B.10. validate.minsize**

| Name | @validate.minsize |
|---|---|
| Description | Validates if a number >= minsize value |
| Types | Integer, Long, int, long |
| **Properties** | |
| value | Required, the minimum size (including) |
| errorcode | Optional |

**Table B.11. validate.pattern**

| Name | @validate.pattern |
|---|---|
| Description | Validates if a String conforms the pattern |
| Types | String |
| **Properties** | |
| value | Required, pattern (requlr expression) to which the string must match |
| errorcode | Optional |

# Appendix C. JSF tags