

Validation Plugin

- [What is Griffon Validation](#)
- [Project Page](#)
- [Dependencies](#)
- [Installation & Upgrade](#)
- [Report a Bug](#)
- [How does GValidation work?](#)
 - [First: Declare Constraints on your Model](#)
 - [Second: Perform Validation in your Controller](#)
- [Error Messages](#)
 - [Model Specific Error Message Code](#)
 - [Default Error Message Code](#)
 - [Real Time Validation Support](#)
- [Validators](#)
 - [blank](#)
 - [creditCard](#)
 - [email](#)
 - [inetAddress](#)
 - [inList](#)
 - [matches](#)
 - [maxSize](#)
 - [max](#)
 - [minSize](#)
 - [min](#)
 - [notEqual](#)
 - [nullable](#)
 - [range](#)
 - [size](#)
 - [url](#)
 - [validator](#)
- [Constraint Inheritance](#)
- [Custom Constraint](#)
- [Perform Validation](#)
 - [Validate All](#)
 - [Selective Validation](#)
 - [Before Validation Callback](#)
- [GValidation Errors](#)
 - [Common Validation Error Usage](#)
 - [Detect Validation Errors](#)
 - [Reject Object or Field](#)
 - [Binding Errors](#)
- [i18n](#)
- [Enhance POGO with Validation Support](#)
 - [Use @Validatable AST Transformation](#)
- [Error Renderer](#)
 - [I. Highlight Error Renderer](#)
 - [II. Popup Error Renderer](#)
 - [III. On With Error Renderer](#)

- [Error Widget](#)
 - [1. Error Messages Widgets](#)
 - [2. Error Icon](#)

What is Griffon Validation

GValidation is a validation plugin for [Griffon](#) - A Grails like application framework for developing desktop applications in Groovy. Like most part of the Griffon framework GValidation's syntax and usage closely resemble its cousin Grails' validation and constraints support. However the similarity pretty much stops here, since Griffon model are very different from Grails' concept of model. In Grails model usually refers to domain classes that are bound to database, however in Griffon models are usually just mere POGOs thus this plugin is designed to work with Griffon models as well as any POGOs.

GValidation is written purely in Groovy while retaining most of the syntax of Grails constraints support.

Project Page

[GitHub Project Page](#)

Dependencies

GValidation plugin depends on the following libraries, and will automatically add them to your application once the plugin is installed.

```
Apache Commons Lang 2.5
Apache Commons Validator 1.3.1
Jakarta ORO 2.0.8
```

Installation & Upgrade

```
griffon install-plugin validation
```

Report a Bug

Please use the [Bug Tracker](#) to report any bug you find

How does GValidation work?

Once this plugin is installed, an additional annotation `@Validatable` will become available to you. Once annotated your model object will have a dynamic field **errors** and a dynamic method **validate** injected similar to Grails domain class. The **errors** field encapsulates all errors generated on a particular model, and the **validate** method performs validation based on the constraints you configure. Here is a typical usage scenario:

First: Declare Constraints on your Model

```
@Validatable
class PersonModel {
    @Bindable String name
    @Bindable String email
    @Bindable String blog

    static constraints = {
        name(blank: false)
        email(blank: false, email: true)
        blog(url: true)
    }
}
```

Second: Perform Validation in your Controller

```
if (!model.validate()) {
    doLater {
        // display error messages
        ...
    }
} else {
    doLater {
        // do the real job
        ...
    }
}
```

Error Messages

Model Specific Error Message Code

All built-in and custom validators provided in this plugin follow the same error message code naming convention. Model specific error message code is generated with the following format:

```
<modelClass>.<field>.<validator>.message
```

So as in the previous example the blank constraint on email field will generate error message code:

```
personModel.email.blank.message
```

Default Error Message Code

Each validator built-in or custom also has a global default error message code associated with in the following format:

```
default.<validator>.message
```

So as shown in the previous example you can provide a global error message for blank validator by using:

```
default.blank.message
```

You can then retrieve the error code and default error code from the Error object using the following fields respectively:

```
error.errorCode  
error.defaultErrorCode
```

Real Time Validation Support

Since v0.8 release validation plugin now can automatically trigger validation for Griffon model beans if a realTime flag is set to true in @Validatable annotation.

```
@Validatable(realTime=true)
class MyModel{
    . . . .
}
```

The real time validation feature is implemented relying on the property change support therefore any property value change, for example triggered by `bind()`, will invoke the validation logic associated with that particular property. The actual validation is performed in a separate thread using `GriffonApplication.execAsync()` before it switches back to EDT for error rendering. Currently this flag only works with Griffon MVC model beans if it is applied to a regular POGO object the flag will be ignored.

Validators

GValidation plugin is shipped with a set of built-in validator or constraints as shown in the above example.

blank

Check if the given String field is blank.

Example:

```
name(blank: false)
```

creditCard

Check if the given field is a valid credit card number.

Exmple:

```
creditCardNumber(creditCard: true)
```

email

Check if the given field is a valid email address.

Example:

```
email(email: true)
```

inetAddress

Check if the given field is a valid network address. It can be either a host name or an IP address. (This is GValidation specific constraint not available in Grails.)

Example:

```
hostServer(inetAddress: true)
```

inList

Check if the given field is contained in the defined list.

Example:

```
city(inList: ['New York', 'Toronto',  
'London'])
```

matches

Check if the field matches with the given regular expression.

Example:

```
login(matches: "[a-zA-Z]+")
```

maxSize

Ensures a value's size does not exceed the given maximum value. This constraint works with collection, array, as well as string.

Example:

```
children(maxSize:25)
firstName(maxSize:20)
```

max

Ensures a value does not exceed the given maximum value.

Example:

```
age(max:new Date())
price(max:999F)
```

minSize

Ensures a value's size does not fall below the given minimum value.

Example:

```
children(minSize:25)
firstName(minSize:2)
```

min

Ensures a value does not fall below the given minimum value.

Example:

```
age(min:new Date())
price(min:0F)
```

notEqual

Ensures that a property is not equal to the specified value

Examples:

```
login(notEqual: "Bob")
```

nullable

Allows a property to be set to null. By default Grails does not allow null values for properties.

Examples:

```
age(nullable: true)
```

range

Uses a Groovy range to ensure that a property's value occurs within a specified range. Set to a Groovy range which can contain numbers in the form of an `IntRange`, dates or any object that implements `Comparable` and provides `next` and `previous` methods for navigation.

Examples:

```
age(range: 18..65)  
createdOn(range: new Date()-10..new Date())
```

size

Uses a Groovy range to restrict the size of a collection or number or the length of a String. Sets the size of a collection or number property or String length.

Examples:

```
children(size: 5..15)
```

url

To validate that a String value is a valid URL. Set to true if a string value is a URL. Internally uses the `org.apache.commons.validator.UrlValidator` class.

Examples:

```
homePage(url:true)
```

validator

Adds custom validation to a field. Set to a closure or block to use for custom validation. A single or no parameter block receives the value, a two-parameter block receives the value and object reference. The closure can return: null or true to indicate that the value is valid false to indicate an invalid value and use the default message code

Examples:

```

// Simple custom validator
even( validator: {
    return (it % 2) == 0
})

// Custom validator with access to the
object under validation
password1( validator: {
    val, obj ->
        obj.properties['password2'] == val
})

// Custom validator with custom error
magicNumber( validator: {
    val, obj ->
        def result = checkMagicNumber()
        if(!result)

obj.errors.rejectValue('magicNumber',
'customErrorCode')
    return result
})

```

Many of the above explanation were borrowed directly from Grails reference guide

Constraint Inheritance

Since the constraints are defined using static fields following Grails convention, no real inheritance can be implemented. However since 0.6 release Validation plugin will basically copy the parent class' constraints to the child before performing validation, thus additionally you can also override the parent constraint in the child class. See the following example:

```

@Validatable
class ServerParameter {
    @Bindable String serverName
    @Bindable int port
    @Bindable String displayName

    def beforeValidation = {
        setDisplayName
"$${serverName}:${port}"
    }

    static constraints = {
        serverName(nullable: false, blank:
false)
        port(range: 0..65535)
        displayName(nullable: false, blank:
false)
    }
}

```

```

@Validatable
class ProtocolSpecificServerParameter
extends ServerParameter{
    @Bindable String protocol

    def beforeValidation = {
        setDisplayName
"$${protocol}://${serverName}:${port}"
    }
}

```

```
static constraints = {  
    protocol(blank: false, nullable:
```

```
false)
    }
}
```

In the above example, the `ProtocolSpecificServerParameter` will not only inherit `ServerParameter`'s `serverName` and `port` fields but also their associated constraints. The only restriction you need to be aware of is if the parent constraint generates error for a certain condition then the overriding child constraint has to generate error as well. In other words, validation plugin does not allow error-hiding by using constraint override in the child class, similar to the method exception treatment during inheritance within Java.

Custom Constraint

The validator mentioned above allows you to specify a closure as a simple custom constraint easily and quickly however there is no easy way to reuse the closure in other scenarios, hence you will be forced to rewrite the validator each time you use them which is inconvenient and a violation of the DRY principle. Since version 0.3, inspired by Grails Custom Constraint plugin, GValidation plugin now provides you ways to define reusable custom constraints in Griffon.

Once you upgrade the plugin to 0.3 version and above a new artifact type will be added to your Griffon application called **Constraint**. You can create new constraints by using the new script added by the plugin:

```
griffon create-constraint
<package>.<constraint-name>
```

Which in turn will create a Groovy class under `griffon-app/constraints` folder with a single method **validate** defined where you can perform your reusable custom validation logic. A simple custom constraint typically looks like this:

```
class MagicConstraint {
```

```

/**
 * Generated message
 *
 * @param propertyValue value of the
property to be validated
 * @param bean object owner of the
property
 * @param parameter configuration
parameter for the constraint
 * @return true if validation passes
otherwise false
 */
def validate(propertyValue, bean,
parameter) {
    if (!parameter)
        return true

    return propertyValue == 42
}
}

```

Once created a custom constraint pretty much behaves exactly like a built-in constraint, you can easily invoke them in your model by following the simple naming convention, following the above example you can apply the constraint on any field in your model by using the following declaration:

```

@Validatable
class DemoModel{
    ...

    @Bindable int magicNumber

    static constraints = {
        ...
        magicNumber(magic: true)
        ...
    }
}

```

The GValidation plugin will also take care of the error message code generation for you, as soon as your validate method returns false the plugin will automatically generate error for the appropriate field with the error message code generated by following the same convention as the built-in ones:

```

<modelClass>.<field>.<validator>.message //
specific error message code
default.<validator>.message // default
global error message code

```

Perform Validation

Validate All

As mentioned before once installed the plugin will enhance all your model object to have the additional **validate()** method, once invoked this method will perform a validate-all executing all constraints on the given model, typical usage scenario:

```
model.validate()  
...  
if(model.hasErrors()){  
    // notify user  
    ...  
}
```

Selective Validation

Originally proposed by Andres Almiray, since v0.3 GValidation now offers capability to perform validation on only a selected number of fields in the model instead of all. Here is a typical single field validation usage scenario:

```
model.validate('name')  
...  
if(model.hasErrors()){  
    // notify user  
    ...  
}
```

Here is how to perform validation on a multiple fields:

```
model.validate(['name', 'email'])  
...  
if(model.hasErrors()){  
    // notify user  
    ...  
}
```

Before Validation Callback

Inspired by Rails `before_validation` callback, now GValidation provides a similar pre-validation callback to give model developer a chance to manipulate data right before validation. Here is an example how this kind of callback is

defined:

```
class ServerModel {
    @Bindable String serverName
    @Bindable int port
    @Bindable String stringForm

    def beforeValidation = {
        setStringForm
        "${serverName}:${port}"
    }

    ...
}
```

GValidation Errors

Although GValidation is not built on top of Spring Validation framework as the Grails constraints do, it still tries to maintain some sort of API consistency when it comes to error generation. However GValidation only provide a subset implementation of the Spring Error objects in Groovy. For API details please see the [Errors](#) and [Simple Error](#) classes.

Common Validation Error Usage

Detect Validation Errors

Other than the return value of the validation() method itself, once the validation is complete you can use the **hasErrors()** method that was dynamically injected on your model to check if there is any validation error. For example:

```
model.validate()
// do something else
...
if(model.hasErrors()){
    notifyUser(model.errors)
}
```

Reject Object or Field

You can generate global error at the instance level by calling the **reject()** method on the **errors** property.

```
model.errors.reject('error.code')
```

Or reject a specific field using the **rejectValue()** method.

```
model.errors.rejectValue('field',  
'error.code')
```

Later on you can iterate through **errors** using Groovy iterator

```
model.errors.each{error->  
    // do something with the error  
}
```

Binding Errors

Since 0.4 release the dynamic **errors** field has been enhanced to be Bindable which means you can now directly bind it to your component. It is especially handy when building error notification component such as in the built-in ErrorMessagePanel. Here is how the binding can be achieved in the view with the built-in panel:

```
container(new  
ErrorMessagePanel(messageSource),  
                id: 'errorMessagePanel',  
constraints: NORTH,  
                errors: bind(source:  
model, 'errors'))
```

i18n

GValidation is shipped with a simple generic errorMessages widget to help you display error message easily. Of

course you can build your own error message feedback component, it is fairly easy to do that, check out the source code for the built-in ErrorMessagePanel for more details. To use the built-in error panel first declare it in your view:

The following example works with v0.4+ binary, if you are using the older version you need to update the errors manually in ErrorMessagePanel

```
panel(id:'demoPanel'){
    BorderLayout()
    errorMessages(constraints: NORTH,
errors: bind(source: model, 'errors'))

    // the rest of your view
}
```

Later in your controller you can update the error messages:

```
def doSomething = {evt = null ->
    if (!model.validate()) {
        doLater {
            // do something
        }
    } else {
        doOutside {
            // do something interesting
        }
    }
}
```

Enhance POGO with Validation Support

Use @Validatable AST Transformation

Since 0.4 release now you can enhance any POGO class in your application by adding the @Validatable annotation at the class level, then Groovy AST transformation will take care of the rest.

```

import
net.sourceforge.gvalidation.annotation.Valid
atable

@Validatable
class AnnotatedModel {
    String id
    String email = " "

    static constraints = {
        id(nullable: false)
        email(email: true)
    }
}

```

This kind of annotated classes will go through essentially the same enhancement as any model class. The only difference is that annotated class is enhanced during build time using AST transformation vs. runtime enhancement as what happen to the model instances.

Error Renderer

One of the common challenge we face when building UI using any GUI framework is how to effectively and easily notify the user about errors. Ideally a validation framework should not just help developer define constraints and validation logic but also handle the presentation of the error message automatically with little coding involved. With this vision in mind Error Renderer was created.

Error Renderer can be declared easily by using the additional synthetic attribute 'errorRenderer' introduced in 0.7 release. See the following example:

```

textField(text: bind(target: model,
    'email'), errorRenderer:'for: email, styles:
[highlight, popup]')

```

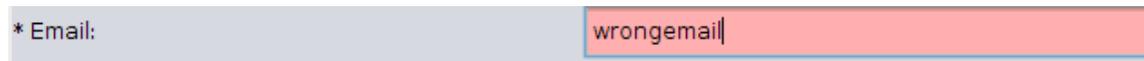
In the above example, two error renderers were declared for the textField widget for the 'email' field in the model. Basically what it means is that if any error was detected for the email field in the model two types of error renderer

will be activated to display the error(s). The styles portion of the configuration is optional. If no renderer style is defined, by default highlight renderer will be used. Currently three types of error renderer styles are implemented, I will go through them quickly here.

I. Highlight Error Renderer

This renderer basically change the background color of the component to pink. Mostly it is used for text based input fields. Here is a screen shot of the rendering result.

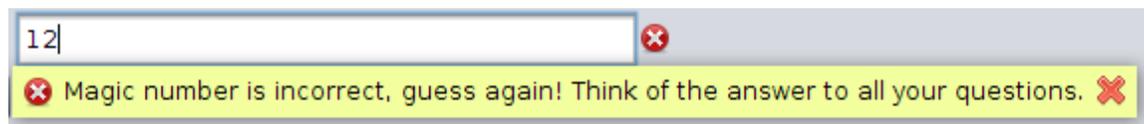
-



II. Popup Error Renderer

This renderer display the error message associated with the error using a tooltip-like popup box. Here is a screen shot of the rendering result.

-



III. On With Error Renderer

This is an invisible renderer that does not render anything itself but switch the component visible attribute on when the error is detected. It is commonly used to display initially invisible custom component when error occurs. This renderer is used in combination of the new errorIcon widget also introduced in this release. Here is a screen shot of it used with errorIcon.

-



Error Widget

1. Error Messages Widgets

This is basically the old wine in a new bottle. This widget is essentially identical to the ErrorMessagePanel class existed since v0.2 however it is now implemented as a widget to make it easier to use. Usage:

```
errorMessages(constraints: NORTH, errors:  
bind(source: model, 'errors'))
```

Screen Shot:

- ✘ Property [email] of class [DemoModel] with value [wrongemail] is not a valid e-mail address
- ✘ Magic number is incorrect, guess again! Think of the answer to all your questions.
- ✘ Property [webSite] of class [DemoModel] with value [www] is not a valid URL

2. Error Icon

As mentioned before this widget is mainly used in combination with the `onWithError` renderer. This icon widget is initially invisible and will only be turned on by the `onWithError` renderer. Usage:

```
errorIcon(errorRenderer: 'for: creditCard,  
styles: [onWithError]')
```

Screen Shot: A screenshot of a web browser address bar. The text 'Web Site (i.e. http://www.yoursite.com/):' is on the left, and 'www' is in the input field on the right. A red error icon is visible in the top right corner of the input field.