Prev Next

Chapter 3. Creating custom constraints

Though the Bean Validation API defines a whole set of standard constraint annotations one can easily think of situations in which these standard annotations won't suffice. For these cases you are able to create custom constraints tailored to your specific validation requirements in a simple manner.

3.1. Creating a simple constraint

To create a custom constraint, the following three steps are required:

- Create a constraint annotation
- Implement a validator
- Define a default error message

3.1.1. The constraint annotation

Let's write a constraint annotation, that can be used to express that a given string shall either be upper case or lower case. We'll apply it later on to the licensePlate field of the Car class from Chapter 1, *Getting started* to ensure, that the field is always an upper-case string.

First we need a way to express the two case modes. We might use String constants, but a better way to go is to use a Java 5 enum for that purpose:

Example 3.1. Enum CaseMode to express upper vs. lower case

```
package com.mycompany;

public enum CaseMode {
    UPPER,
    LOWER;
}
```

Now we can define the actual constraint annotation. If you've never designed an annotation before, this may look a bit scary, but actually it's not that hard:

Example 3.2. Defining CheckCase constraint annotation

```
package com.mycompany;

import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.*;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import javax.validation.Constraint;
import javax.validation.Payload;

@Target( { METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Constraint(validatedBy = CheckCaseValidator.class)
@Documented
public @interface CheckCase {

    String message() default "{com.mycompany.constraints.checkcase}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    CaseMode value();

}
```

An annotation type is defined using the @interface keyword. All attributes of an annotation type are declared in a method-like manner. The specification of the Bean Validation API demands, that any constraint annotation defines

- an attribute message that returns the default key for creating error messages in case the constraint is violated

- an attribute groups that allows the specification of validation groups, to which this constraint belongs (see Section 2.3, “Validating groups”). This must default to an empty array of type Class<?>.
- an attribute payload that can be used by clients of the Bean Validation API to assign custom payload objects to a constraint. This attribute is not used by the API itself.

> ### Tip
>
> An example for a custom payload could be the definition of a severity.
>
> ```
> public class Severity {
>     public static class Info extends Payload {};
>     public static class Error extends Payload {};
> }
>
> public class ContactDetails {
>     @NotNull(message="Name is mandatory", payload=Severity.Error.class)
>     private String name;
>
>     @NotNull(message="Phone number not specified, but not mandatory", payload=Severity.Info.class)
>     private String phoneNumber;
>
>     // ...
> }
> ```
>
> Now a client can after the validation of a ContactDetails instance access the severity of a constraint using ConstraintViolation.getConstraintDescriptor().getPayload() and adjust its behaviour depending on the severity.

Besides those three mandatory attributes (message, groups and payload) we add another one allowing for the required case mode to be specified. The name value is a special one, which can be omitted upon using the annotation, if it is the only attribute specified, as e.g. in @CheckCase(CaseMode.UPPER).

In addition we annotate the annotation type with a couple of so-called meta annotations:

- @Target({ METHOD, FIELD, ANNOTATION_TYPE }): Says, that methods, fields and annotation declarations may be annotated with @CheckCase (but not type declarations e.g.)
- @Retention(RUNTIME): Specifies, that annotations of this type will be available at runtime by the means of reflection
- @Constraint(validatedBy = CheckCaseValidator.class): Specifies the validator to be used to validate elements annotated with @CheckCase
- @Documented: Says, that the use of @CheckCase will be contained in the JavaDoc of elements annotated with it

3.1.2. The constraint validator

Next, we need to implement a constraint validator, that's able to validate elements with a @CheckCase annotation. To do so, we implement the interface ConstraintValidator as shown below:

Example 3.3. Implementing a constraint validator for the constraint CheckCase

```
package com.mycompany;

import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;

public class CheckCaseValidator implements ConstraintValidator<CheckCase, String> {

    private CaseMode caseMode;

    public void initialize(CheckCase constraintAnnotation) {
        this.caseMode = constraintAnnotation.value();
    }

    public boolean isValid(String object, ConstraintValidatorContext constraintContext) {

        if (object == null)
            return true;

        if (caseMode == CaseMode.UPPER)
            return object.equals(object.toUpperCase());
        else
            return object.equals(object.toLowerCase());
    }

}
```

The ConstraintValidator interface defines two type parameters, which we set in our implementation. The first one specifies the annotation type to be validated (in our example CheckCase), the second one the type of elements, which the validator can handle (here String).

In case a constraint annotation is allowed at elements of different types, a ConstraintValidator for each allowed type has to be implemented and registered at the constraint annotation as shown above.

The implementation of the validator is straightforward. The initialize() method gives us access to the attribute values of the annotation to be validated. In the example we store the CaseMode in a field of the validator for further usage.

In the isValid() method we implement the logic, that determines, whether a String is valid according to a given @CheckCase annotation or not. This decision depends on the case mode retrieved in initialize(). As the Bean Validation specification recommends, we consider null values as being valid. If null is not a valid value for an element, it should be annotated with @NotNull explicitly.

3.1.2.1. The ConstraintValidatorContext

Example 3.3, "Implementing a constraint validator for the constraint CheckCase" relies on the default error message generation by just returning true or false from the isValid call. Using the passed ConstraintValidatorContext object it is possible to either add additional error messages or completely disable the default error message generation and solely define custom error messages. The ConstraintValidatorContext API is modeled as fluent interface and is best demonstrated with an example:

Example 3.4. Use of ConstraintValidatorContext to define custom error messages

```
package com.mycompany;

import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;

public class CheckCaseValidator implements ConstraintValidator<CheckCase, String> {

    private CaseMode caseMode;

    public void initialize(CheckCase constraintAnnotation) {
        this.caseMode = constraintAnnotation.value();
    }

    public boolean isValid(String object, ConstraintValidatorContext constraintContext) {

        if (object == null)
            return true;

        boolean isValid;
        if (caseMode == CaseMode.UPPER) {
            isValid = object.equals(object.toUpperCase());
        }
        else {
            isValid = object.equals(object.toLowerCase());
        }

        if(!isValid) {
            constraintContext.disableDefaultConstraintViolation();
            constraintContext.buildConstraintViolationWithTemplate( "{com.mycompany.constraints.CheckCase
        }
        return result;
    }

}
```

Example 3.4, "Use of ConstraintValidatorContext to define custom error messages" shows how you can disable the default error message generation and add a custom error message using a specified message template. In this example the use of the ConstraintValidatorContext results in the same error message as the default error message generation.

Tip

It is important to end each new constraint violation with addConstraintViolation. Only after that the new constraint violation will be created.

In case you are implementing a ConstraintValidator a class level constraint it is also possible to adjust set the property path for the created constraint violations. This is important for the case where you validate multiple properties of the class or even traverse the object graph. A custom property path creation could look like Example 3.5, “Adding new ConstraintViolation with custom property path”.

Example 3.5. Adding new ConstraintViolation with custom property path

```
public boolean isValid(Group group, ConstraintValidatorContext constraintValidatorContext) {
    boolean isValid = false;
    ...

    if(!isValid) {
        constraintValidatorContext
            .buildConstraintViolationWithTemplate( "{my.custom.template}" )
            .addNode( "myProperty" ).addConstraintViolation();
    }
    return isValid;
}
```

3.1.3. The error message

Finally we need to specify the error message, that shall be used, in case a @CheckCase constraint is violated. To do so, we add the following to our custom **ValidationMessages.properties** (see also Section 2.2.4, “Message interpolation”)

Example 3.6. Defining a custom error message for the CheckCase constraint

```
com.mycompany.constraints.CheckCase.message=Case mode must be {value}.
```

If a validation error occurs, the validation runtime will use the default value, that we specified for the message attribute of the @CheckCase annotation to look up the error message in this file.

3.1.4. Using the constraint

Now that our first custom constraint is completed, we can use it in the Car class from the Chapter 1, *Getting started* chapter to specify that the licensePlate field shall only contain upper-case strings:

Example 3.7. Applying the CheckCase constraint

```
package com.mycompany;

import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

public class Car {

    @NotNull
    private String manufacturer;

    @NotNull
    @Size(min = 2, max = 14)
    @CheckCase(CaseMode.UPPER)
    private String licensePlate;

    @Min(2)
    private int seatCount;
    public Car(String manufacturer, String licencePlate, int seatCount) {

        this.manufacturer = manufacturer;
        this.licensePlate = licencePlate;
        this.seatCount = seatCount;
    }

    //getters and setters ...

}
```

Finally let's demonstrate in a little test that the @CheckCase constraint is properly validated:

Example 3.8. Testcase demonstrating the CheckCase validation

```
package com.mycompany;

import static org.junit.Assert.*;

import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

import org.junit.BeforeClass;
import org.junit.Test;

public class CarTest {

    private static Validator validator;

    @BeforeClass
    public static void setUp() {
        ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
        validator = factory.getValidator();
    }

    @Test
    public void testLicensePlateNotUpperCase() {

        Car car = new Car("Morris", "dd-ab-123", 4);

        Set<ConstraintViolation<Car>> constraintViolations =
            validator.validate(car);
        assertEquals(1, constraintViolations.size());
        assertEquals(
            "Case mode must be UPPER.",
            constraintViolations.iterator().next().getMessage());
    }

    @Test
    public void carIsValid() {

        Car car = new Car("Morris", "DD-AB-123", 4);

        Set<ConstraintViolation<Car>> constraintViolations =
            validator.validate(car);

        assertEquals(0, constraintViolations.size());
    }
}
```

3.2. Constraint composition

Looking at the licensePlate field of the Car class in Example 3.7, “Applying the CheckCase constraint”, we see three constraint annotations already. In complexer scenarios, where even more constraints could be applied to one element, this might become a bit confusing easily. Furthermore, if we had a licensePlate field in another class, we would have to copy all constraint declarations to the other class as well, violating the DRY principle.

This problem can be tackled using compound constraints. In the following we create a new constraint annotation @ValidLicensePlate, that comprises the constraints @NotNull, @Size and @CheckCase:

Example 3.9. Creating a composing constraint ValidLicensePlate

```
package com.mycompany;

import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.*;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import javax.validation.Constraint;
import javax.validation.Payload;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

@NotNull
@Size(min = 2, max = 14)
@CheckCase(CaseMode.UPPER)
@Target( { METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Constraint(validatedBy = {})
@Documented
public @interface ValidLicensePlate {

    String message() default "{com.mycompany.constraints.validlicenseplate}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

}
```

To do so, we just have to annotate the constraint declaration with its comprising constraints (btw. that's exactly why we allowed annotation types as target for the @CheckCase annotation). As no additional validation is required for the @ValidLicensePlate annotation itself, we don't declare a validator within the @Constraint meta annotation.

Using the new compound constraint at the licensePlate field now is fully equivalent to the previous version, where we declared the three constraints directly at the field itself:

Example 3.10. Application of composing constraint ValidLicensePlate

```
package com.mycompany;

public class Car {

    @ValidLicensePlate
    private String licensePlate;

    //...

}
```

The set of ConstraintViolations retrieved when validating a Car instance will contain an entry for each violated composing constraint of the @ValidLicensePlate constraint. If you rather prefer a single ConstraintViolation in case any of the composing constraints is violated, the @ReportAsSingleViolation meta constraint can be used as follows:

Example 3.11. Usage of @ReportAsSingleViolation

```
//...
@ReportAsSingleViolation
public @interface ValidLicensePlate {

    String message() default "{com.mycompany.constraints.validlicenseplate}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

}
```

Prev Top of page Front page Next