Hibernate Validator

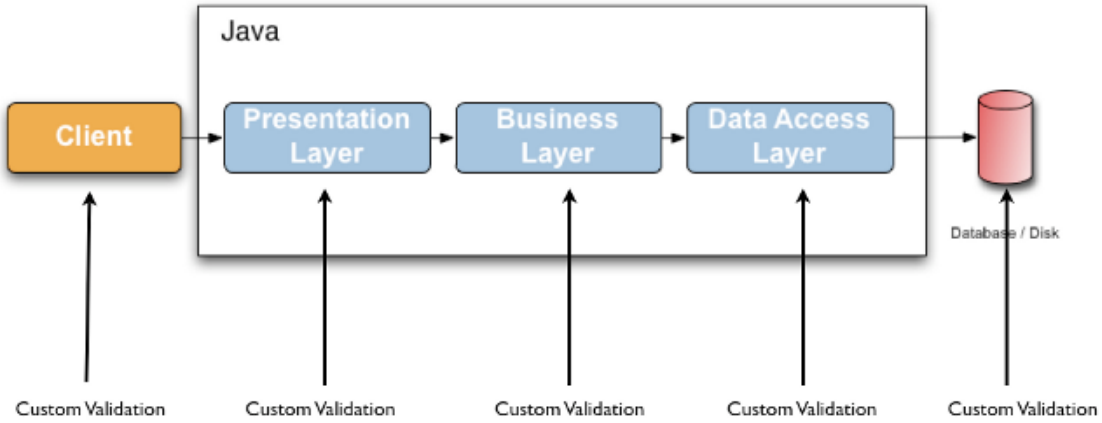
JSR 303 Reference Implementation

Reference Guide

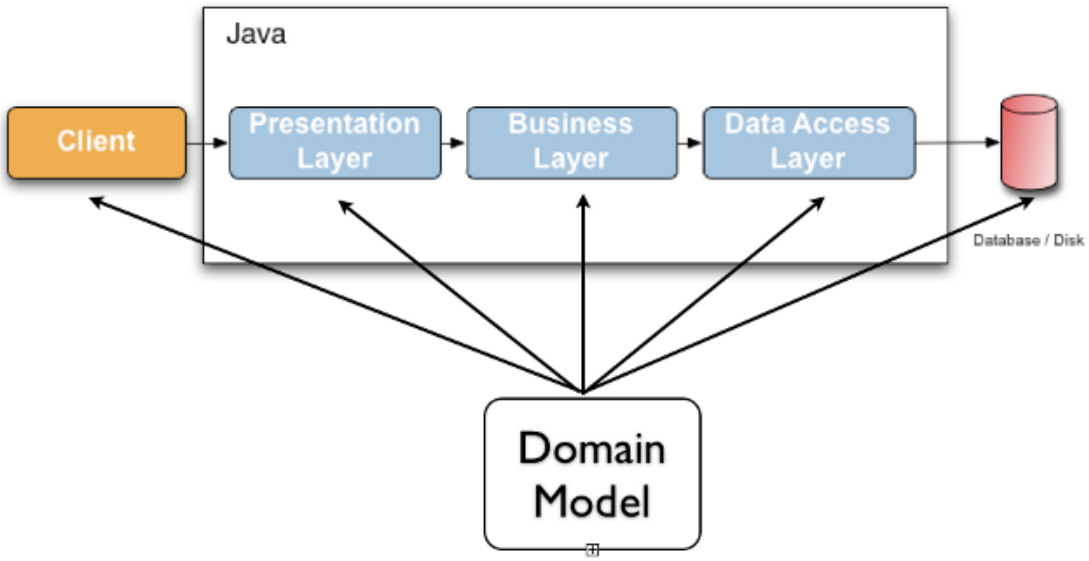
4.0.1.GA

Preface

Validating data is a common task that occurs throughout any application, from the presentation layer to the persistence layer. Often the same validation logic is implemented in each layer, proving time consuming and error-prone. To avoid duplication of these validations in each layer, developers often bundle validation logic directly into the domain model, cluttering domain classes with validation code which is really metadata about the class itself.



JSR 303 - Bean Validation - defines a metadata model and API for entity validation. The default metadata source is annotations, with the ability to override and extend the meta-data through the use of XML. The API is not tied to a specific application tier or programming model. It is specifically not tied to either the web tier or the persistence tier, and is available for both server-side application programming, as well as rich client Swing application developers.



Hibernate Validator is the reference implementation of this JSR.

Getting started

This chapter will show you how to get started with Hibernate Validator, the reference implementation (RI) of Bean Validation. For the following quickstart you need:

- A JDK >= 5
- *Apache Maven* [http://maven.apache.org/]
- An Internet connection (Maven has to download all required libraries)
- A properly configured remote repository. Add the following to your `settings.xml`:

Example 1.1. Configuring the JBoss Maven repository in `settings.xml`

```
<repositories>
   <repository>
      <id>jboss</id>
      <url>http://repository.jboss.com/maven2</url>
      <releases>
         <enabled>true</enabled>
      </releases>
      <snapshots>
         <enabled>false</enabled>
      </snapshots>
   </repository>
</repositories>
```

More information about `settings.xml` can be found in the *Maven Local Settings Model* [http://maven.apache.org/ref/2.0.8/maven-settings/settings.html].

1.1. Setting up a new Maven project

Start by creating new Maven project using the Maven archetype plugin as follows:

Example 1.2. Using Maven's archetype plugin to create a sample project using Hibernate Validator

```
mvn archetype:generate \
  -DarchetypeCatalog=http://repository.jboss.com/maven2/archetype-catalog.xml \
  -DgroupId=com.mycompany \
  -DartifactId=beanvalidation-gettingstarted \
  -Dversion=1.0-SNAPSHOT \
```

```
-Dpackage=com.mycompany
```

When presented with the list of available archetypes in the JBoss Maven Repository select the *hibernate-validator-quickstart-archetype.* After Maven has downloaded all dependencies confirm the settings by just pressing enter. Maven will create your project in the directory `beanvalidation-gettingstarted`. Change into this directory and run:

```
mvn test
```

Maven will compile the example code and run the implemented unit tests. Let's have a look at the actual code.

1.2. Applying constraints

Open the project in the IDE of your choice and have a look at the class `Car`:

Example 1.3. Class Car annotated with constraints

```
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)
   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 ...
}
```

`@NotNull`, `@Size` and `@Min` are so-called constraint annotations, that we use to declare constraints, which shall be applied to the fields of a `Car` instance:

- manufacturer shall never be null
- licensePlate shall never be null and must be between 2 and 14 characters long
- seatCount shall be at least 2.

1.3. Validating constraints

To perform a validation of these constraints, we use a `Validator` instance. Let's have a look at the `CarTest` class:

Example 1.4. Class CarTest showing validation examples

```
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 manufacturerIsNull() {
```

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

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

        assertEquals(1, constraintViolations.size());
        assertEquals("may not be null", constraintViolations.iterator().next().getMessage());
    }

    @Test
    public void licensePlateTooShort() {
        Car car = new Car("Morris", "D", 4);

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

        assertEquals(1, constraintViolations.size());
                                    assertEquals("size must be between 2 and 14",
constraintViolations.iterator().next().getMessage());
    }

    @Test
    public void seatCountTooLow() {
        Car car = new Car("Morris", "DD-AB-123", 1);

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

        assertEquals(1, constraintViolations.size());
                                  assertEquals("must be greater than or equal to 2",
constraintViolations.iterator().next().getMessage());
    }

    @Test
    public void carIsValid() {
        Car car = new Car("Morris", "DD-AB-123", 2);

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

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

In the `setUp()` method we get a `Validator` instance from the `ValidatorFactory`. A `Validator` instance is thread-safe and may be reused multiple times. For this reason we store it as field of our test class. We can use the `Validator` now to validate the different car instances in the test methods.

The `validate()` method returns a set of `ConstraintViolation` instances, which we can iterate in order to see which validation errors occurred. The first three test methods show some expected constraint violations:

- The `@NotNull` constraint on manufacturer is violated in `manufacturerIsNull()`
- The `@Size` constraint on licensePlate is violated in `licensePlateTooShort()`
- The `@Min` constraint on seatCount is violated in `seatCountTooLow()`

If the object validates successfully, `validate()` returns an empty set.

Note that we only use classes from the package javax.validation from the Bean Validation API. As we don't reference any classes of the RI directly, it would be no problem to switch to another implementation of the API, should that need arise.

1.4. Where to go next?

That concludes our 5 minute tour through the world of Hibernate Validator. Continue exploring the code or look at further examples referenced in *Chapter 7, Further reading*. To get a deeper understanding of the Bean Validation just continue reading.*Chapter 2, Validation step by step*. In case your application has specific validation requirements have a look at *Chapter 3, Creating custom constraints*.

Validation step by step

In this chapter we will see in more detail how to use Hibernate Validator to validate constraints for a given entity model. We will also learn which default constraints the Bean Validation specification provides and which additional constraints are only provided by Hibernate Validator. Let's start with how to add constraints to an entity.

2.1. Defining constraints

Constraints in Bean Validation are expressed via Java annotations. In this section we show how to annotate an object model with these annotations. We have to differentiate between three different type of constraint annotations - field-, property-, and class-level annotations.

Note

Not all constraints can be placed on all of these levels. In fact, none of the default constraints defined by Bean Validation can be placed at class level. The `java.lang.annotation.Target` annotation in the constraint annotation itself determines on which elements a constraint can be placed. See *Chapter 3, Creating custom constraints* for more information.

2.1.1. Field-level constraints

Constraints can be expressed by annotating a field of a class. *Example 2.1, "Field level constraint"* shows a field level configuration example:

Example 2.1. Field level constraint

```
package com.mycompany;

import javax.validation.constraints.NotNull;

public class Car {

    @NotNull
    private String manufacturer;

    @AssertTrue
    private boolean isRegistered;

    public Car(String manufacturer, boolean isRegistered) {
        super();
```

```
        this.manufacturer = manufacturer;
        this.isRegistered = isRegistered;
    }
}
```

When using field level constraints field access strategy is used to access the value to be validated. This means the instance variable directly independed of the access type.

Note

The access type (private, protected or public) does not matter.

Note

Static fields and properties cannot be validated.

2.1.2. Property-level constraints

If your model class adheres to the *JavaBeans* [http://java.sun.com/javase/technologies/desktop/javabeans/index.jsp] standard, it is also possible to annotate the properties of a bean class instead of its fields. *Example 2.2, "Property level constraint"* uses the same entity as in *Example 2.1, "Field level constraint"*, however, property level constraints are used.

Note

The property's getter method has to be annotated, not its setter.

Example 2.2. Property level constraint

```
package com.mycompany;

import javax.validation.constraints.AssertTrue;
import javax.validation.constraints.NotNull;

public class Car {

    private String manufacturer;

    private boolean isRegistered;
```

```
    public Car(String manufacturer, boolean isRegistered) {
        super();
        this.manufacturer = manufacturer;
        this.isRegistered = isRegistered;
    }

    @NotNull
    public String getManufacturer() {
        return manufacturer;
    }

    public void setManufacturer(String manufacturer) {
        this.manufacturer = manufacturer;
    }

    @AssertTrue
    public boolean isRegistered() {
        return isRegistered;
    }

    public void setRegistered(boolean isRegistered) {
        this.isRegistered = isRegistered;
    }
}
```

When using property level constraints property access strategy is used to access the value to be validated. This means the bean validation provider accesses the state via the property accessor method.

Tip

It is recommended to stick either to field *or* property annotation within one class. It is not recommended to annotate a field *and* the accompanying getter method as this would cause the field to be validated twice.

2.1.3. Class-level constraints

Last but not least, a constraint can also be placed on class level. When a constraint annotation is placed on this level the class instance itself passed to the `ConstraintValidator`. Class level constraints are useful if it is necessary to inspect more than a single property of the class to validate it or if a correlation between different state variables has to be evaluated. In *Example 2.3, "Class level constraint"* we add the property passengers to the class `Car`. We also add the constraint `PassengerCount` on the class level. We will later see how we can actually create this custom

constraint (see *Chapter 3, Creating custom constraints*). For now we it is enough to know that `PassengerCount` will ensure that there cannot be more passengers in a car than there are seats.

Example 2.3. Class level constraint

```
package com.mycompany;

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

@PassengerCount
public class Car {

    @NotNull
    private String manufacturer;

    @NotNull
    @Size(min = 2, max = 14)
    private String licensePlate;

    @Min(2)
    private int seatCount;

    private List<Person> passengers;

    public Car(String manufacturer, String licencePlate, int seatCount) {
        this.manufacturer = manufacturer;
        this.licensePlate = licencePlate;
        this.seatCount = seatCount;
    }

    //getters and setters ...
}
```

2.1.4. Constraint inheritance

When validating an object that implements an interface or extends another class, all constraint annotations on the implemented interface and parent class apply in the same manner as the constraints specified on the validated object itself. To make things clearer let's have a look at the following example:

Example 2.4. Constraint inheritance using RentalCar

```
package com.mycompany;

import javax.validation.constraints.NotNull;

public class RentalCar extends Car {

    private String rentalStation;

    public RentalCar(String manufacturer, String rentalStation) {
        super(manufacturer);
        this.rentalStation = rentalStation;
    }

    @NotNull
    public String getRentalStation() {
        return rentalStation;
    }

    public void setRentalStation(String rentalStation) {
        this.rentalStation = rentalStation;
    }
}
```

Our well-known class `Car` from *???* is now extended by `RentalCar` with the additional property rentalStation. If an instance of `RentalCar` is validated, not only the `@NotNull` constraint on rentalStation is validated, but also the constraint on manufacturer from the parent class.

The same would hold true, if `Car` were an interface implemented by `RentalCar`.

Constraint annotations are aggregated if methods are overridden. If `RentalCar` would override the `getManufacturer()` method from `Car` any constraints annotated at the overriding method would be evaluated in addition to the `@NotNull` constraint from the super-class.

2.1.5. Object graphs

The Bean Validation API does not only allow to validate single class instances but also complete object graphs. To do so, just annotate a field or property representing a reference to another object with `@Valid`. If the parent object is validated, all referenced objects annotated with `@Valid` will be validated as well (as will be their children etc.). See *Example 2.6, “Adding a driver to the car”*.

Example 2.5. Class Person

```
package com.mycompany;

import javax.validation.constraints.NotNull;

public class Person {

   @NotNull
   private String name;

   public Person(String name) {
      super();
      this.name = name;
   }

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }
}
```

Example 2.6. Adding a driver to the car

```
package com.mycompany;

import javax.validation.Valid;
import javax.validation.constraints.NotNull;

public class Car {

   @NotNull
   @Valid
   private Person driver;

   public Car(Person driver) {
      this.driver = driver;
   }
```

```
    //getters and setters ...
}
```

If an instance of `Car` is validated, the referenced `Person` object will be validated as well, as the driver field is annotated with `@Valid`. Therefore the validation of a `Car` will fail if the name field of the referenced `Person` instance is `null`.

Object graph validation also works for collection-typed fields. That means any attributes that are

- arrays
- implement `java.lang.Iterable` (especially `Collection`, `List` and `Set`)
- implement `java.util.Map`

can be annotated with `@Valid`, which will cause each contained element to be validated, when the parent object is validated.

Example 2.7. Car with a list of passengers

```
package com.mycompany;

import java.util.ArrayList;
import java.util.List;

import javax.validation.Valid;
import javax.validation.constraints.NotNull;

public class Car {

    @NotNull
    @Valid
    private List<Person> passengers = new ArrayList<Person>();

    public Car(List<Person> passengers) {
        this.passengers = passengers;
    }

    //getters and setters ...
}
```

If a `Car` instance is validated, a `ConstraintValidation` will be created, if any of the `Person` objects contained in the passengers list has a `null` name.

Note

`null` values are getting ignored when validating object graphs.

2.2. Validating constraints

The `Validator` interface is the main entry point to Bean Validation. In *Section 5.1, "Configuration and ValidatorFactory"* we will first show how to obtain an `Validator` instance. Afterwards we will learn how to use the different methods of the `Validator` interface.

2.2.1. Obtaining a `Validator` instance

The first step towards validating an entity instance is to get hold of a `Validator` instance. The road to this instance leads via the `Validation` class and a `ValidatorFactory`. The easiest way is to use the static `Validation.buildDefaultValidatorFactory()` method:

Example 2.8. Validation.buildDefaultValidatorFactory()

```
ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
Validator validator = factory.getValidator();
```

For other ways of obtaining a Validator instance see *Chapter 5, Bootstrapping*. For now we just want to see how we can use the `Validator` instance to validate entity instances.

2.2.2. Validator methods

The `Validator` interface contains three methods that can be used to either validate entire entities or just a single properties of the entity.

All three methods return a `Set<ConstraintViolation>`. The set is empty, if the validation succeeds. Otherwise a `ConstraintViolation` instance is added for each violated constraint.

All the validation methods have a var-args parameter which can be used to specify, which validation groups shall be considered when performing the validation. If the parameter is not specified the default validation group (`javax.validation.Default`) will be used. We will go into more detail on the topic of validation groups in *Section 2.3, "Validating groups"*

2.2.2.1. `validate`

Use the `validate()` method to perform validation of all constraints of a given entity instance (see *Example 2.9, "Usage of Validator.validate()"*).

Example 2.9. Usage of `Validator.validate()`

```
ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
Validator validator = factory.getValidator();

Car car = new Car(null);

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

assertEquals(1, constraintViolations.size());
assertEquals("may not be null", constraintViolations.iterator().next().getMessage());
```

2.2.2.2. `validateProperty`

With help of the `validateProperty()` a single named property of a given object can be validated. The property name is the JavaBeans property name.

Example 2.10. Usage of `Validator.validateProperty()`

```
Validator validator = Validation.buildDefaultValidatorFactory().getValidator();

Car car = new Car(null);

Set<ConstraintViolation<Car>> constraintViolations = validator.validateProperty(car,
 "manufacturer");

assertEquals(1, constraintViolations.size());
assertEquals("may not be null", constraintViolations.iterator().next().getMessage());
```

`Validator.validateProperty` is for example used in the integration of Bean Validation into JSF 2 (see *Section 6.3, "Presentation layer validation"*).

2.2.2.3. `validateValue`

Using the `validateValue()` method you can check, whether a single property of a given class can be validated successfully, if the property had the specified value:

Example 2.11. Usage of `Validator.validateValue()`

```
Validator validator = Validation.buildDefaultValidatorFactory().getValidator();
```

```
Set<ConstraintViolation<Car>> constraintViolations = validator.validateValue(Car.class,
 "manufacturer", null);

assertEquals(1, constraintViolations.size());
assertEquals("may not be null", constraintViolations.iterator().next().getMessage());
```

Note

`@Valid` is not honored by `validateProperty()` or `validateValue()`.

2.2.3. `ConstraintViolation` methods

Now it is time to have a closer look at what a `ConstraintViolation`. Using the different methods of `ConstraintViolation` a lot of useful information about the cause of the validation failure can be determined. *Table 2.1, "The various ConstraintViolation methods"* gives an overview of these methods:

Table 2.1. The various `ConstraintViolation` methods

Method	Usage	Example (refering to *Example 2.9, "Usage of Validator.validate()"*)
`getMessage()`	The interpolated error message.	may not be null
`getMessageTemplate()`	The non-interpolated error message.	{javax.validation.constraints.NotNull.message}
`getRootBean()`	The root bean being validated.	car
`getRootBeanClass()`	The class of the root bean being validated.	Car.class
`getLeafBean()`	If a bean constraint, the bean instance the constraint is applied on. If a property constraint, the bean instance hosting the property the constraint is applied on.	car
`getPropertyPath()`	The property path to the value from root bean.	
`getInvalidValue()`	The value failing to pass the constraint.	passengers
`getConstraintDescriptor()`	Constraint metadata reported to fail.	

2.2.4. Message interpolation

As we will see in *Chapter 3, Creating custom constraints* each constraint definition must define a default message descriptor. This message can be overridden at declaration time using the `message` attribute of the constraint. You can see this in *Example 2.13, "Driver"*. This message descriptors get interpolated when a constraint validation fails using the configured `MessageInterpolator`. The interpolator will try to resolve any message parameters, meaning string literals enclosed in braces. In order to resolve these parameters Hibernate Validator's default `MessageInterpolator` first recursively resolves parameters against a custom `ResourceBundle` called `ValidationMessages.properties` at the root of the classpath (It is up to you to create this file). If no further replacements are possible against the custom bundle the default `ResourceBundle` under `/org/hibernate/validator/ValidationMessages.properties` gets evaluated. If a replacement occurs against the default bundle the algorithm looks again at the custom bundle (and so on). Once no further replacements against these two resource bundles are possible remaining parameters are getting resolved against the attributes of the constraint to be validated.

Since the braces { and } have special meaning in the messages they need to be escaped if they are used literally. The following The following rules apply:

- \{ is considered as the literal {
- \} is considered as the literal }
- \\ is considered as the literal \

If the default message interpolator does not fit your requirements it is possible to plug a custom `MessageInterpolator` when the `ValidatorFactory` gets created. This can be seen in *Chapter 5, Bootstrapping*.

2.3. Validating groups

Groups allow you to restrict the set of constraints applied during validation. This makes for example wizard like validation possible where in each step only a specified subset of constraints get validated. The groups targeted are passed as var-args parameters to `validate`, `validateProperty` and `validateValue`. Let's have a look at an extended `Car` with `Driver` example. First we have the class `Person` (*Example 2.12, "Person"*) which has a `@NotNull` constraint on name. Since no group is specified for this annotation its default group is `javax.validation.Default`.

Note

When more than one group is requested, the order in which the groups are evaluated is not deterministic. If no group is specified the default group `javax.validation.Default` is assumed.

Example 2.12. Person

```
public class Person {
   @NotNull
   private String name;

   public Person(String name) {
      this.name = name;
   }
   // getters and setters ...
}
```

Next we have the class `Driver` (*Example 2.13, "Driver"*) extending `Person`. Here we are adding the properties age and hasDrivingLicense. In order to drive you must be at least 18 (`@Min(18)`) and you must have a driving license (`@AssertTrue`). Both constraints defined on these properties belong to the group `DriverChecks`. As you can see in *Example 2.14, "Group interfaces"* the group `DriverChecks` is just a simple tagging interface. Using interfaces makes the usage of groups type safe and allows for easy refactoring. It also means that groups can inherit from each other via class inheritance.

Note

The Bean Validation specification does not enforce that groups have to be interfaces. Non interface classes could be used as well, but we recommend to stick to interfaces.

Example 2.13. Driver

```
public class Driver extends Person {
      @Min(value = 18, message = "You have to be 18 to drive a car", groups =
DriverChecks.class)
   public int age;

      @AssertTrue(message = "You first have to pass the driving test", groups =
DriverChecks.class)
   public boolean hasDrivingLicense;

   public Driver(String name) {
      super( name );
   }
```

```
    public void passedDrivingTest(boolean b) {
        hasDrivingLicense = b;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

Example 2.14. Group interfaces

```
public interface DriverChecks {
}

public interface CarChecks {
}
```

Last but not least we add the property passedVehicleInspection to the `Car` class (*Example 2.15, "Car"*) indicating whether a car passed the road worthy tests.

Example 2.15. Car

```
public class Car {
    @NotNull
    private String manufacturer;

    @NotNull
    @Size(min = 2, max = 14)
    private String licensePlate;

    @Min(2)
    private int seatCount;

     @AssertTrue(message = "The car has to pass the vehicle inspection first", groups =
 CarChecks.class)
    private boolean passedVehicleInspection;

    @Valid
```

```
    private Driver driver;

    public Car(String manufacturer, String licencePlate, int seatCount) {
        this.manufacturer = manufacturer;
        this.licensePlate = licencePlate;
        this.seatCount = seatCount;
    }
}
```

Overall three different groups are used in our example. Person.name, Car.manufacturer, Car.licensePlate and Car.seatCount all belong to the `Default` group. Driver.age and Driver.hasDrivingLicense belong to `DriverChecks` and last but not least Car.passedVehicleInspection belongs to the group `CarChecks`. *Example 2.16, "Drive away"* shows how passing different group combinations to the `Validator.validate` method result in different validation results.

Example 2.16. Drive away

```
public class GroupTest {

    private static Validator validator;

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

    @Test
    public void driveAway() {
        // create a car and check that everything is ok with it.
        Car car = new Car( "Morris", "DD-AB-123", 2 );
        Set<ConstraintViolation<Car>> constraintViolations = validator.validate( car );
        assertEquals( 0, constraintViolations.size() );

        // but has it passed the vehicle inspection?
        constraintViolations = validator.validate( car, CarChecks.class );
        assertEquals( 1, constraintViolations.size() );
                    assertEquals("The car has to pass the vehicle inspection first",
 constraintViolations.iterator().next().getMessage());

        // let's go to the vehicle inspection
        car.setPassedVehicleInspection( true );
```

```
        assertEquals( 0, validator.validate( car ).size() );

        // now let's add a driver. He is 18, but has not passed the driving test yet
        Driver john = new Driver( "John Doe" );
        john.setAge( 18 );
        car.setDriver( john );
        constraintViolations = validator.validate( car, DriverChecks.class );
        assertEquals( 1, constraintViolations.size() );
                        assertEquals( "You first have to pass the driving test",
 constraintViolations.iterator().next().getMessage() );

        // ok, John passes the test
        john.passedDrivingTest( true );
        assertEquals( 0, validator.validate( car, DriverChecks.class ).size() );

        // just checking that everything is in order now
                    assertEquals( 0, validator.validate( car, Default.class, CarChecks.class,
 DriverChecks.class ).size() );
    }
}
```

First we create a car and validate it using no explicit group. There are no validation errors, even though the property passedVehicleInspection is per default `false`. However, the constraint defined on this property does not belong to the default group.

Next we just validate the `CarChecks` group which will fail until we make sure that the car passes the vehicle inspection.

When we then add a driver to the car and validate against `DriverChecks` we get again a constraint violation due to the fact that the driver has not yet passed the driving test. Only after setting passedDrivingTest to true the validation against `DriverChecks` will pass.

Last but not least, we show that all constraints are passing by validating against all defined groups.

2.3.1. Group sequences

By default, constraints are evaluated in no particular order and this regardless of which groups they belong to. In some situations, however, it is useful to control the order of the constraints evaluation. In our example from *Section 2.3, "Validating groups"* we could for example require that first all default car constraints are passing before we check the road worthiness of the car. Finally before we drive away we check the actual driver constraints. In order to implement such an order one would define a new interface and annotate it with `@GroupSequence` defining the order in which the groups have to be validated.

Note

If at least one constraints fails in a sequenced group none of the constraints of the follwoing groups in the sequence get validated.

Example 2.17. Interface with @GroupSequence

```
@GroupSequence({Default.class, CarChecks.class, DriverChecks.class})
public interface OrderedChecks {
}
```

Warning

Groups defining a sequence and groups composing a sequence must not be involved in a cyclic dependency either directly or indirectly, either through cascaded sequence definition or group inheritance. If a group containing such a circularity is evaluated, a `GroupDefinitionException` is raised.

The usage of the new sequence could then look like in *Example 2.18, "Usage of a group sequence"*.

Example 2.18. Usage of a group sequence

```
@Test
public void testOrderedChecks() {
    Car car = new Car( "Morris", "DD-AB-123", 2 );
    car.setPassedVehicleInspection( true );

    Driver john = new Driver( "John Doe" );
    john.setAge( 18 );
    john.passedDrivingTest( true );
    car.setDriver( john );

    assertEquals( 0, validator.validate( car, OrderedChecks.class ).size() );
}
```

2.3.2. Redefining the default group sequence of a class

The @GroupSequence annotation also fulfills a second purpose. It allows you to redefine what the Default group means for a given class. To redefine `Default` for a class, place a `@GroupSequence` annotation on the class. The defined groups in the annotation express the sequence of groups that substitute `Default` for this class. *Example 2.19, "RentalCar"* introduces a new class RentalCar with a redfined default group. With this definition the check for all three groups can be rewritten as seen in *Example 2.20, "testOrderedChecksWithRedefinedDefault"*.

Example 2.19. RentalCar

```
@GroupSequence({ RentalCar.class, CarChecks.class })
public class RentalCar extends Car {
   public RentalCar(String manufacturer, String licencePlate, int seatCount) {
      super( manufacturer, licencePlate, seatCount );
   }
}
```

Example 2.20. testOrderedChecksWithRedefinedDefault

```
@Test
public void testOrderedChecksWithRedefinedDefault() {
   RentalCar rentalCar = new RentalCar( "Morris", "DD-AB-123", 2 );
   rentalCar.setPassedVehicleInspection( true );

   Driver john = new Driver( "John Doe" );
   john.setAge( 18 );
   john.passedDrivingTest( true );
   rentalCar.setDriver( john );

   assertEquals( 0, validator.validate( rentalCar, Default.class, DriverChecks.class ).size() );
}
```

Note

Due to the fact that there cannot be a cyclic dependency in the group and group sequence definitions one cannot just add `Default` to the sequence redefining `Default` for a class. Instead the class itself should be added!

2.4. Built-in constraints

Hibernate Validator implements all of the default constraints specified in Bean Validation as well as some custom ones. *Table 2.2, "Built-in constraints"* list all constraints available in Hibernate Validator.

Table 2.2. Built-in constraints

Annotation	Part of Bean Validation Specification	Apply on	Use	Hibernate Metadata impact
@AssertFalse	yes	field/property	check that the annotated element is `false`.	none
@AssertTrue	yes	field/property	check that the annotated element is `true`.	none
@DecimalMax	yes	field/property. Supported types are `BigDecimal`, `BigInteger`, `String`, `byte`, `short`, `int`, `long` and the respective wrappers of the primitive types.	The annotated element must be a number whose value must be lower or equal to the specified maximum. The parameter value is the string representation of the max value according to the `BigDecimal` string representation.	none
@DecimalMin	yes	field/property. Supported types are `BigDecimal`, `BigInteger`, `String`, `byte`, `short`, `int`, `long` and the respective wrappers of the primitive types.	The annotated element must be a number whose value must be higher or equal to the specified minimum. The parameter value is the string representation of the min value according to	none

Annotation	Part of Bean Validation Specification	Apply on	Use	Hibernate Metadata impact
			the `BigDecimal` string representation.	
@Digits(integer=, fraction=)	yes	field/property. Supported types are `BigDecimal`, `BigInteger`, `String`, `byte`, `short`, `int`, `long` and the respective wrappers of the primitive types.	Check whether the property is a number having up to `integer` digits and `fraction` fractional digits.	Define column precision and scale.
@Email	no	field/property. Needs to be a string.	Check whether the specified string is a valid email address.	none
@Future	yes	field/property. Supported types are `java.util.Date` and `java.util.Calendar`.	Checks whether the annotated date is in the future.	none
@Length(min=, max=)	no	field/property. Needs to be a string.	Validate that the annotated string is between *`min`* and *`max`* included.	none
@Max	yes	field/property. Supported types are `BigDecimal`, `BigInteger`, `String`, `byte`, `short`, `int`, `long` and the respective wrappers of the primitive types.	Checks whether the annotated value is less than or equal to the specified maximum.	Add a check constraint on the column.

Annotation	Part of Bean Validation Specification	Apply on	Use	Hibernate Metadata impact
@Min	yes	field/property. Supported types are `BigDecimal`, `BigInteger`, `String`, `byte`, `short`, `int`, `long` and the respective wrappers of the primitive types.	Check whether the annotated value is higher than or equal to the specified minimum.	Add a check constraint on the column.
@NotNull	yes	field/property	Check that the annotated value is not `null`.	Column(s) are not null.
@NotEmpty	no	field/property. Needs to be a string.	Check if the string is not `null` nor empty.	none
@Null	yes	field/property	Check that the annotated value is `null`.	none
@Past	yes	field/property. Supported types are `java.util.Date` and `java.util.Calendar`.	Checks whether the annotated date is in the past.	none
@Pattern(regex=, flag=)	yes	field/property. Needs to be a string.	Check if the annotated string match the regular expression *`regex`*.	none
@Range(min=, max=)	no	field/property. Supported types are `BigDecimal`, `BigInteger`, `String`, `byte`, `short`, `int`, `long` and the respective	Check whether the annotated value lies between (inclusive) the specified minimum and maximum.	none

Annotation	Part of Bean Validation Specification	Apply on	Use	Hibernate Metadata impact
		wrappers of the primitive types.		
@Size(min=, max=)	yes	field/property. Supported types are `String`, `Collection`, `Map` and `arrays`.	Check if the annotated element size is between min and max (inclusive).	Column length will be set to max.
@Valid	yes	field/property	Perform validation recursively on the associated object.	none

Note

On top of the parameters indicated in *Table 2.2, "Built-in constraints"* each constraint supports the parameters `message`, `groups` and `payload`. This is a requirement of the Bean Validation specification.

In some cases these built-in constraints will not fulfill your requirements. In this case you can literally in a minute write your own constraints. We will discuss this in *Chapter 3, Creating custom constraints*

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.ConstraintPayload;

@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 asign custom payload objects to a constraint. This attribute is not used by the API itself.

Tip

An examle for a custom payload could be the definition of a severity.

```
public class Severity {
```

```
    public static class Info extends ConstraintPayload {};
    public static class Error extends ConstraintPayload {};
}

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 (messge, 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` explicitely.

The passed-in `ConstraintValidatorContext` could be used to raise any custom validation errors, but as we are fine with the default behavior, we can ignore that parameter for now.

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 3.4. 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 3.5. 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 3.6. 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 3.5, "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 3.7. 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.ConstraintPayload;
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 3.8. 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 3.9. Usage of `@ReportAsSingleViolation`

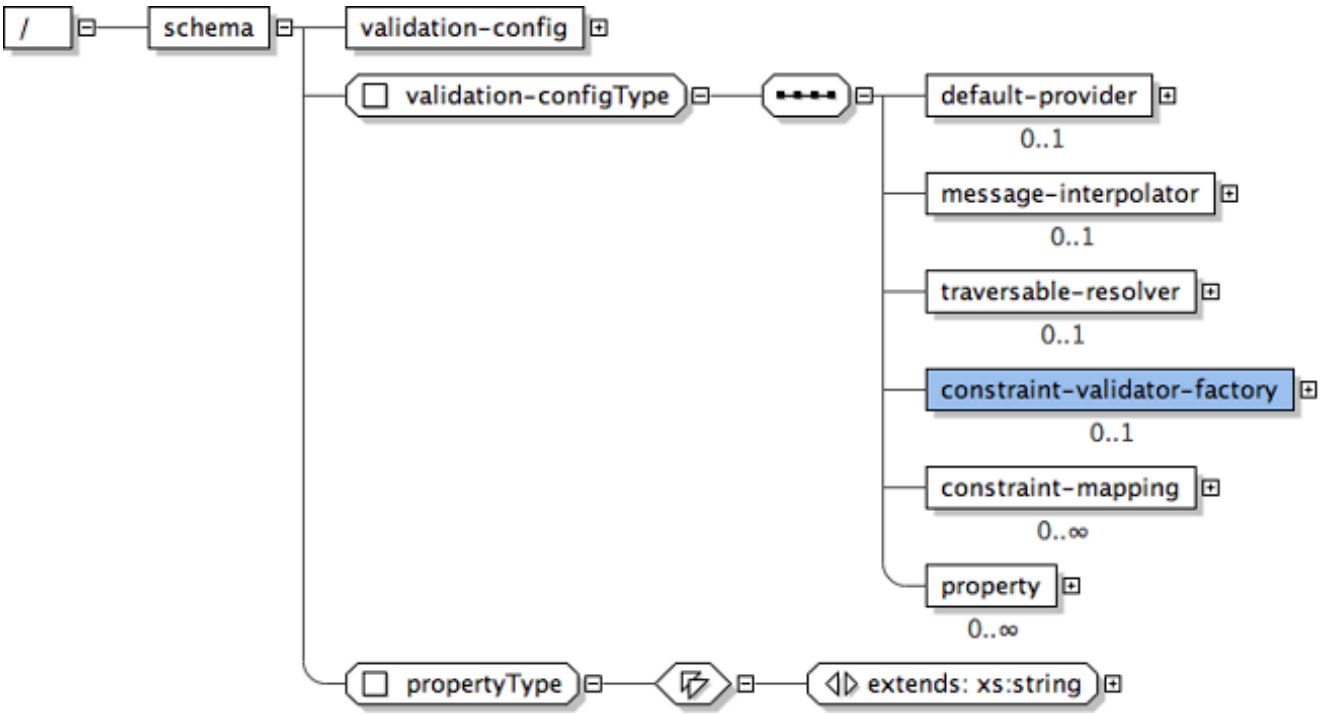
```
//...
@ReportAsSingleViolation
public @interface ValidLicensePlate {

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

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

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

}
```

XML configuration

4.1. `validation.xml`

The key to enable XML configuration for Hibernate Validator is the file `validation.xml`. If this file exists in the classpath its configuration will be applied when the `ValidationFactory` gets created. *Example 4.1, "validation-configuration-1.0.xsd"* shows a model view of the xsd `valiation.xml` has to adhere to.

Example 4.1. validation-configuration-1.0.xsd



Example 4.2, "validation.xml" shows the several configuration options of `validation.xml`.

Example 4.2. validation.xml

```
<validation-config xmlns="http://jboss.org/xml/ns/javax/validation/configuration"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://jboss.org/xml/ns/javax/validation/configuration">
    <default-provider>org.hibernate.validator.HibernateValidator</default-provider>
    <message-interpolator>org.hibernate.validator.engine.ResourceBundleMessageInterpolator</
message-interpolator>
      <traversable-resolver>org.hibernate.validator.engine.resolver.DefaultTraversableResolver</
traversable-resolver>
     <constraint-validator-factory>org.hibernate.validator.engine.ConstraintValidatorFactoryImpl</
constraint-validator-factory>
```

```
    <constraint-mapping>/constraints-car.xml</constraint-mapping>
    <property name="prop1">value1</property>
    <property name="prop2">value2</property>
</validation-config>
```

Warning

There can only be one `validation.xml` in the classpath. If more than one is found an exception is thrown.

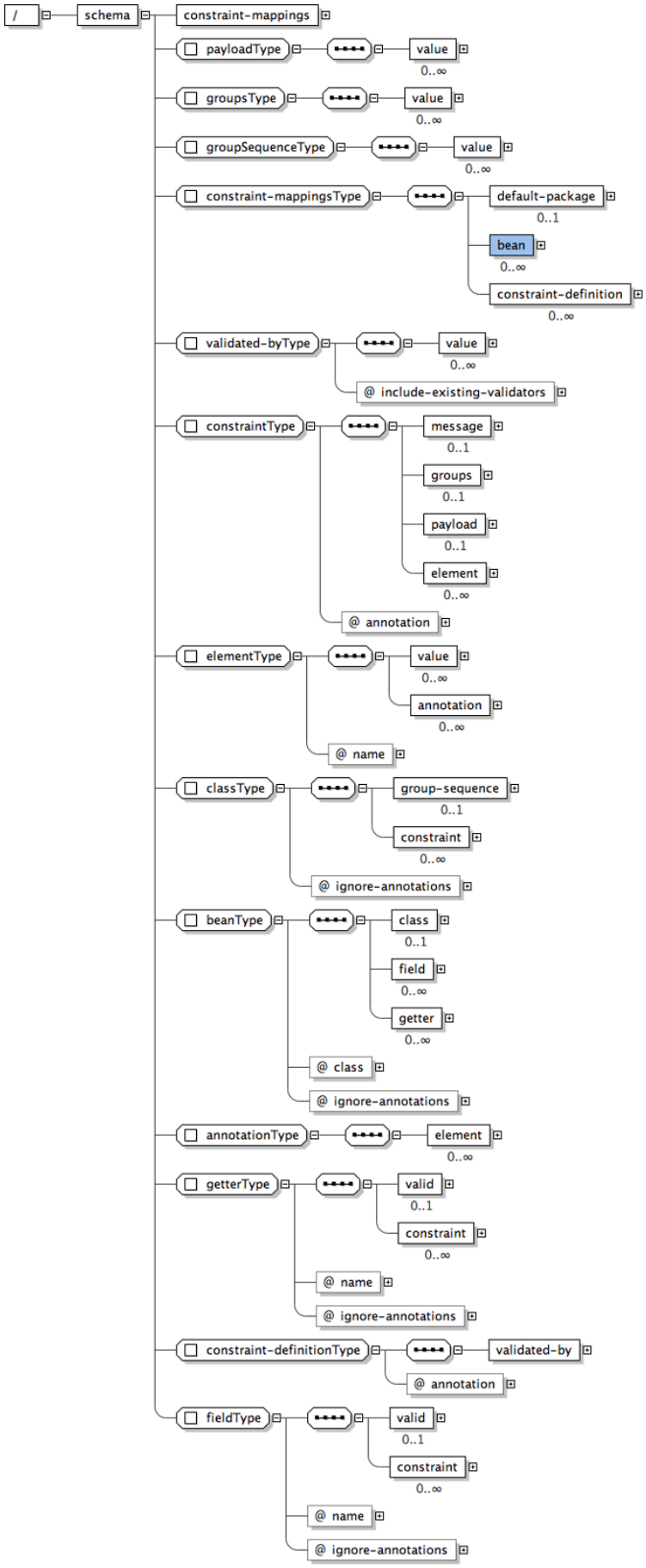
All settings shown in the `validation.xml` are optional and in the case of *Example 4.2, "validation.xml"* show the defaults used within Hibernate Validator. The node default-provider allows to choose the Bean Validation provider. This is useful if there is more than one provider in the classpath. message-interpolator, traversable-resolver and constraint-validator-factory allow to customize the `javax.validation.MessageInterpolator`, `javax.validation.TraversableResolver` resp. `javax.validation.ConstraintValidatorFactory`. The same configuration options are also available programmatically through the `javax.validation.Configuration`. In fact XML configuration will be overriden by values explicitly specified via the API. It is even possible to ignore the XML configuration completely via `Configuration.ignoreXmlConfiguration()`. See also *Chapter 5, Bootstrapping*.

Via the constraint-mapping you can list an arbitrary number of additional XML files containing the actual constraint configuration. See *Section 4.2, "Mapping constraints"*.

Last but not least, you can specify provider specific properties via the property nodes. Hibernate Validator does currently not make use of any custom properties.

4.2. Mapping constraints

Expressing constraints in XML is possible via files adhering to the xsd seen in *Example 4.3, "validation-mapping-1.0.xsd"*. Note that these mapping files are only processed if listed via constraint-mapping in your `validation.xml`.


/
schema
constraint-mappings
payloadType
value
0..∞
groupsType
value
0..∞
groupSequenceType
value
0..∞
constraint-mappingsType
default-package
0..1
bean
0..∞
constraint-definition
0..∞
validated-byType
value
0..∞
@ include-existing-validators
constraintType
message
0..1
groups
0..1
payload
0..1
element
0..∞
@ annotation
elementType
value
0..∞
annotation
0..∞
@ name
classType
group-sequence
0..1
constraint
0..∞
@ ignore-annotations
beanType
class
0..1
field
0..∞
getter
0..∞
@ class
@ ignore-annotations
annotationType
element
0..∞
getterType
valid
0..1
constraint
0..∞
@ name
@ ignore-annotations
constraint-definitionType
validated-by
@ annotation
fieldType
valid
0..1
constraint
0..∞
@ name
@ ignore-annotations

Example 4.4, "constraints-car.xml" shows how our classes Car and RentalCar from *Example 2.15, "Car"* resp. *Example 2.19, "RentalCar"* could be mapped in XML.

Example 4.4. constraints-car.xml

```
<constraint-mappings xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                     xsi:schemaLocation="http://jboss.org/xml/ns/javax/validation/mapping validation-
mapping-1.0.xsd"
                xmlns="http://jboss.org/xml/ns/javax/validation/mapping">
   <default-package>org.hibernate.validator.quickstart</default-package>
   <bean class="Car" ignore-annotations="true">
      <field name="manufacturer">
         <constraint annotation="javax.validation.constraints.NotNull"/>
      </field>
      <field name="licensePlate">
         <constraint annotation="javax.validation.constraints.NotNull"/>
      </field>
      <field name="seatCount">
         <constraint annotation="javax.validation.constraints.Min">
            <element name="value">2</element>
         </constraint>
      </field>
      <field name="driver">
         <valid/>
      </field>
      <getter name="passedVehicleInspection" ignore-annotations="true">
         <constraint annotation="javax.validation.constraints.AssertTrue">
            <message>The car has to pass the vehicle inspection first</message>
            <groups>
               <value>CarChecks</value>
            </groups>
            <element name="max">10</element>
         </constraint>
      </getter>
   </bean>
   <bean class="RentalCar" ignore-annotations="true">
      <class ignore-annotations="true">
         <group-sequence>
            <value>RentalCar</value>
            <value>CarChecks</value>
         </group-sequence>
      </class>
   </bean>
```

```
            <constraint-definition annotation="org.mycompany.CheckCase" include-existing-validator="false">
        <validated-by include-existing-validators="false">
          <value>org.mycompany.CheckCaseValidator</value>
        </validated-by>
    </constraint-definition>
</constraint-mappings>
```

The XML configuration is closely mirroring the programmatic API. For this reason it should suffice to just add some comments. default-package is used for all fields where a classname is expected. If the specified class is not fully qualified the configured default package will be used. Every mapping file can then have several bean nodes, each describing the constraints on the entity with the specified class name.

Warning

A given entity can only be configured once across all configuration files. If the same class is configured more than once an exception is thrown.

Settings ignore-annotations to true means that constraint annotations placed on the configured bean are ignored. The default for this value is `true`. ignore-annotations is also available for the nodes class, fields and getter. If not explicitly specified on these levels the configured bean value applies. Otherwise do the nodes class, fields and getter determine on which level the constraints are placed (see *Section 2.1, "Defining constraints"*). The constraint node is then used to add a constraint on the corresponding level. Each constraint definition must define the class via the annotation attribute. The constraint attributes required by the Bean Validation specification (message, groups and payload) have dedicated nodes. All other constraint specific attributes are configured using the the element node.

The class node also allows to reconfigure the default group sequence (see *Section 2.3.2, "Redefining the default group sequence of a class"*) via the group-sequence node.

Last but not least, the list of `ConstraintValidators` associated to a given constraint can be altered via the constraint-definition node. The annotation attribute represents the constraint annotation being altered. The validated-by elements represent the (ordered) list of `ConstraintValidator` implementations associated to the constraint. If include-existing-validator is set to `false`, validators defined on the constraint annotation are ignored. If set to `true`, the list of ConstraintValidators described in XML are concatenated to the list of validators described on the annotation.

Bootstrapping

We already seen in *Section 5.1, "Configuration and ValidatorFactory"* the easiest way to create a `Validator` instance - `Validation.buildDefaultValidatorFactory`. In this chapter we have a look at the other methods in `javax.validation.Validation` and how they allow to configure several aspects of Bean Validation at bootstrapping time.

The different bootstrapping options allwow, amongst other things, to bootstrap any Bean Validation implementation on the classpath. Generally, an available provider is discovered by the *Java Service Provider* [http://java.sun.com/j2se/1.3/docs/guide/jar/jar.html#Service%20Provider] mechanism. A Bean Validation implementation includes the file `javax.validation.spi.ValidationProvider` in `META-INF/services`. This file contains the fully qualified classname of the `ValidationProvider` of the implementation. In the case of Hibernate Validator this is `org.hibernate.validator.HibernateValidator`.

Note

If there are more than one Bean Validation implementation providers in the classpath and `Validation.buildDefaultValidatorFactory()` is used, there is no guarantee which provider will be chosen. To enforce the provider `Validation.byProvider()` should be used.

5.1. `Configuration` and `ValidatorFactory`

There are three different methods in the Validation class to create a Validator instance. The easiest in shown in *Example 5.1, "Validation.buildDefaultValidatorFactory()"*.

Example 5.1. Validation.buildDefaultValidatorFactory()

```
ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
Validator validator = factory.getValidator();
```

You can also use the method `Validation.byDefaultProvider()` which will allow you to configure several aspects of the created Validator instance:

Example 5.2. Validation.byDefaultProvider()

```
Configuration<?> config = Validation.byDefaultProvider().configure();
config.messageInterpolator(new MyMessageInterpolator())
    .traversableResolver( new MyTraversableResolver())
    .constraintValidatorFactory(new MyConstraintValidatorFactory());
```

```
ValidatorFactory factory = config.buildValidatorFactory();
Validator validator = factory.getValidator();
```

We will learn more about `MessageInterpolator`, `TraversableResolver` and `ConstraintValidatorFactory` in the following sections.

Last but not least you can ask for a Configuration object of a specific Bean Validation provider. This is useful if you have more than one Bean Validation provider in your classpath. In this situation you can make an explicit choice about which implementation to use. In the case of Hibernate Validator the `Validator` creation looks like:

Example 5.3. Validation.byProvider(HibernateValidator.class)

```
ValidatorConfiguration config = Validation.byProvider( HibernateValidator.class ).configure();
config.messageInterpolator(new MyMessageInterpolator())
    .traversableResolver( new MyTraversableResolver())
    .constraintValidatorFactory(new MyConstraintValidatorFactory());

ValidatorFactory factory = config.buildValidatorFactory();
Validator validator = factory.getValidator();
```

Tip

The generated `Validator` instance is thread safe and can be cached.

5.2. `ValidationProviderResolver`

In the case that the Java Service Provider mechanism does not work in your environment or you have a special classloader setup, you are able to provide a custom `ValidationProviderResolver`. An example in an OSGi environment you could plug your custom provider resolver like seen in *Example 5.4, "Providing a custom ValidationProviderResolver"*.

Example 5.4. Providing a custom ValidationProviderResolver

```
Configuration<?> config = Validation.byDefaultProvider()
    .providerResolver( new OSGiServiceDiscoverer() )
    .configure();

ValidatorFactory factory = config.buildValidatorFactory();
```

```
Validator validator = factory.getValidator();
```

Your `OSGiServiceDiscoverer` must in this case implement the interface `ValidationProviderResolver`:

Example 5.5. ValidationProviderResolver interface

```
public interface ValidationProviderResolver {
   /**
    * Returns a list of ValidationProviders available in the runtime environment.
    *
    * @return list of validation providers.
    */
   List<ValidationProvider<?>> getValidationProviders();
}
```

5.3. `MessageInterpolator`

Section 2.2.4, "Message interpolation" already discussed the default message interpolation algorithm. If you have special requirements for your message interpolation you can provide a custom interpolator using `Configuration.messageInterpolator()`. This message interpolator will be shared by all validators generated by the `ValidatorFactory` created from this `Configuration`(see *Example 5.6, "Providing a custom MessageInterpolator"*).

Example 5.6. Providing a custom MessageInterpolator

```
Configuration<?> configuration = Validation.byDefaultProvider().configure();
ValidatorFactory factory = configuration
                                                    .messageInterpolator(new
 ContextualMessageInterpolator(configuration.getDefaultMessageInterpolator()))
   .buildValidatorFactory();

Validator validator = factory.getValidator();
```

Tip

It is recommended that `MessageInterpolator` implementations delegate final interpolation to the Bean Validation default `MessageInterpolator` to ensure standard Bean Validation interpolation

rules are followed. The default implementation is accessible through `Configuration.getDefaultMessageInterpolator()`.

5.4. `TraversableResolver`

The usage of the `TraversableResolver` has so far not been discussed. The idea is that in some cases, the state of a property should not be accessed. The most obvious example for that is a lazy loaded property or association of a Java Persistence provider. Validating this lazy property or association would mean that its state would have to be accessed triggering a load from the database. Bean Validation controls which property can and cannot be accessed via the `TraversableResolver` interface (see *Example 5.7, "TraversableResolver interface"*).

Example 5.7. TraversableResolver interface

```
/**
 * Contract determining if a property can be accessed by the Bean Validation provider
 * This contract is called for each property that is being either validated or cascaded.
 *
 * A traversable resolver implementation must be thread-safe.
 *
 */
public interface TraversableResolver {
    /**
     * Determine if the Bean Validation provider is allowed to reach the property state
     *
     * @param traversableObject object hosting <code>traversableProperty</code> or null
     *                          if validateValue is called
     * @param traversableProperty the traversable property.
     * @param rootBeanType type of the root object passed to the Validator.
     * @param pathToTraversableObject path from the root object to
     *        <code>traversableObject</code>
     *        (using the path specification defined by Bean Validator).
     * @param elementType either <code>FIELD</code> or <code>METHOD</code>.
     *
     * @return <code>true</code> if the Bean Validation provider is allowed to
     *         reach the property state, <code>false</code> otherwise.
     */
    boolean isReachable(Object traversableObject,
                        Path.Node traversableProperty,
                        Class<?> rootBeanType,
                        Path pathToTraversableObject,
                        ElementType elementType);
```

```
    /**
     * Determine if the Bean Validation provider is allowed to cascade validation on
     * the bean instance returned by the property value
     * marked as <code>@Valid</code>.
     * Note that this method is called only if isReachable returns true for the same set of
     * arguments and if the property is marked as <code>@Valid</code>
     *
     * @param traversableObject object hosting <code>traversableProperty</code> or null
     *                          if validateValue is called
     * @param traversableProperty the traversable property.
     * @param rootBeanType type of the root object passed to the Validator.
     * @param pathToTraversableObject path from the root object to
     *        <code>traversableObject</code>
     *        (using the path specification defined by Bean Validator).
     * @param elementType either <code>FIELD</code> or <code>METHOD</code>.
     *
     * @return <code>true</code> if the Bean Validation provider is allowed to
     *         cascade validation, <code>false</code> otherwise.
     */
     boolean isCascadable(Object traversableObject,
                          Path.Node traversableProperty,
                          Class<?> rootBeanType,
                          Path pathToTraversableObject,
                          ElementType elementType);
}
```

Hibernate Validator provides two `TraversableResolvers` out of the box which will be enabled automatically depending on your environment. The first is the `DefaultTraversableResolver` which will always return true for `isReachable()` and `isTraversable()`. The second is the `JPATraversableResolver` which gets enabled when Hibernate Validator gets used in combination with JPA 2. In case you have to provide your own resolver you can do so again using the `Configuration` object as seen in *Example 5.8, "Providing a custom TraversableResolver"*.

Example 5.8. Providing a custom TraversableResolver

```
Configuration<?> configuration = Validation.byDefaultProvider().configure();
ValidatorFactory factory = configuration
   .traversableResolver(new MyTraversableResolver())
   .buildValidatorFactory();

Validator validator = factory.getValidator();
```

5.5. `ConstraintValidatorFactory`

Last but not least, there is one more configuration option to discuss, the `ConstraintValidatorFactory`. The default `ConstraintValidatorFactory` provided by Hibernate Validator requires a public no-arg constructor to instantiate `ConstraintValidator` instances (see *Section 3.1.2, "The constraint validator"*). Using a custom `ConstraintValidatorFactory` offers for example the possibility to use dependency injection in constraint implementations. The configuration of the custom factory is once more via the `Configuration` (*Example 5.9, "Providing a custom ConstraintValidatorFactory"*).

Example 5.9. Providing a custom ConstraintValidatorFactory

```
Configuration<?> configuration = Validation.byDefaultProvider().configure();
ValidatorFactory factory = configuration
    .constraintValidatorFactory(new IOCConstraintValidatorFactory())
    .buildValidatorFactory();

Validator validator = factory.getValidator();
```

The interface you have to implement is:

Example 5.10. ConstraintValidatorFactory interface

```
public interface ConstraintValidatorFactory {
    /**
     * @param key The class of the constraint validator to instantiate.
     *
     * @return A constraint validator instance of the specified class.
     */
     <T extends ConstraintValidator<?,?>> T getInstance(Class<T> key);
}
```

Warning

Any constraint implementation relying on `ConstraintValidatorFactory` behaviors specific to an implementation (dependency injection, no no-arg constructor and so on) are not considered portable.

Note

ConstraintValidatorFactory should not cache instances as the state of each instance can be altered in the initialize method.

Integration with other frameworks

Hibernate Validator is intended to be used to implement multi-layered data validation, where constraints are expressed in a single place (the annotated domain model) and checked in various different layers of the application.

6.1. Database schema-level validation

Out of the box, Hibernate Annotations (as of Hibernate 3.5.x) will translate the constraints you have defined for your entities into mapping metadata. For example, if a property of your entity is annotated `@NotNull`, its columns will be declared as `not null` in the DDL schema generated by Hibernate.

If, for some reason, the feature needs to be disabled, set `hibernate.validator.apply_to_ddl` to `false`. See also *Table 2.2, "Built-in constraints"*.

You can also limit the DDL constraint generation to a subset of the defined constraints by setting the property org.hibernate.validator.group.ddl. The property specifies the comma seperated, fully specified classnames of the groups a constraint has to be part of in order to be considered for DDL schema generation.

6.2. ORM integration

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

6.2.1. Hibernate event-based validation

Hibernate Validator has a built-in Hibernate event listener - *`org.hibernate.cfg.beanvalidation.BeanValidationEventListener`* [http://fisheye.jboss.org/browse/Hibernate/core/trunk/annotations/src/main/java/org/hibernate/cfg/beanvalidation/BeanValidationEventListener.java] - which is part of Hibernate Annotations (as of Hibernate 3.5.x). Whenever a `PreInsertEvent`, `PreUpdateEvent` or `PreDeleteEvent` occurs, the listener will verify all constraints of the entity instance and throw an exception if any constraint is violated. Per default objects will be checked before any inserts or updates are made by Hibernate. Pre deletion events will per default not trigger a validation. You can configure the groups to be validated per event type using the properties javax.persistence.validation.group.pre-persist, javax.persistence.validation.group.pre-update and javax.persistence.validation.group.pre-remove. The values of these properties are the comma seperated, fully specified class names of the groups to validate. *Example 6.1, "Manual configuration of BeanValidationEvenListener"* shows the default values for these properties. In this case they could also be omitted.

On constraint violation, the event will raise a runtime `ConstraintViolationException` which contains a set of `ConstraintViolations` describing each failure.

If Hibernate Validator is present in the classpath, Hibernate Annotations (or Hibernate EntityManager) will use it transparently. To avoid validation even though Hibernate Validator is in the classpath set javax.persistence.validation.mode to `none`.

Note

If the beans are not annotated with validation annotations, there is no runtime performance cost.

In case you need to manually set the event listeners for Hibernate Core, use the following configuration in `hibernate.cfg.xml`:

Example 6.1. Manual configuration of `BeanValidationEvenListener`

```
<hibernate-configuration>
   <session-factory>
      ...
       <property name="javax.persistence.validation.group.pre-persist">javax.validation.Default</
property>
       <property name="javax.persistence.validation.group.pre-update">javax.validation.Default</
property>
       <property name="javax.persistence.validation.group.pre-remove"></property>
   </session-factory>
   <event type="pre-update">
    <listener class="org.hibernate.cfg.beanvalidation.BeanValidationEventListener"/>
   </event>
   <event type="pre-insert">
    <listener class="org.hibernate.cfg.beanvalidation.BeanValidationEventListener"/>
   </event>
   <event type="pre-delete">
    <listener class="org.hibernate.cfg.beanvalidation.BeanValidationEventListener"/>
   </event>
</hibernate-configuration>
```

6.2.2. JPA

If you are using JPA 2 and Hibernate Validator is in the classpath the JPA2 specification requires that Bean Validation gets enabled. The properties javax.persistence.validation.group.pre-persist, javax.persistence.validation.group.pre-update and javax.persistence.validation.group.pre-remove as described in *Section 6.2.1, "Hibernate event-based validation"* can in this case be configured in `persistence.xml`. `persistence.xml` also defines a node validation-mode while can be set to `AUTO`, `CALLBACK`, `NONE`. The default is `AUTO`.

In a JPA 1 you will have to create and register Hibernate Validator yourself. In case you are using Hibernate EntityManager you can add a customized version of the `BeanValidationEventListener` described in *Section 6.2.1, "Hibernate event-based validation"* to your project and register it manually.

6.3. Presentation layer validation

When working with JSF2 or JBoss Seam and Hibernate Validator (Bean Validation) is present in the runtime environment validation is triggered for every field in the application. *???* shows an example of the f:validateBean tag in a JSF page. For more information refer to the Seam documentation or the JSF 2 specification.

Example 6.2. Usage of Bean Validation within JSF2

```
<h:form>
  <f:validateBean>
    <h:inputText value="#{model.property}" />
    <h:selectOneRadio value="#{model.radioProperty}" > ... </h:selectOneRadio>
    <!-- other input components here -->
  </f:validateBean>
</h:form>
```

Further reading

Last but not least, a few pointers to further information. A great source for examples is the Bean Validation TCK which can is available for anonymous access in the Hibernate *SVN repository* [http://anonsvn.jboss.org/repos/hibernate/validator/trunk]. Alternatively you can view the tests using *Hibernate's fisheye* [http://fisheye.jboss.org/browse/Hibernate/beanvalidation/trunk/validation-tck/src/main/java/org/hibernate/jsr303/tck/tests] installation. *The JSR 303* [http://jcp.org/en/jsr/detail?id=303] specification itself is also a great way to deepen your understanding of Bean Validation resp. Hibernate Validator.

If you have any furhter questions to Hibernate Validator or want to share some of your use cases have a look at the *Hibernate Validator Wiki* [http://www.hibernate.org/469.html] and the *Hibernate Validator Forum* [https://forum.hibernate.org/viewforum.php?f=9].

In case you would like to report a bug use *Hibernate's Jira* [http://opensource.atlassian.com/projects/hibernate/browse/HV] instance. Feedback is always welcome!