Bean Validation

jim stafford

Fall 2024 v2021-05-13: Built: 2024-12-03 22:28 EST

Table of Contents

Chapter 1. Introduction

Well-designed software components should always be designed according to a contract of what is required of inputs and outputs; constraints; or pre-conditions and post-conditions. Validation of inputs and outputs need to be performed at component boundaries. These conditions need to be well-advertised, but ideally the checking of these conditions should not overwhelm the functional aspects of the code.

Manual Validation

```
public PersonPocDTO createPOC(PersonPocDTO personDTO) {
    if (null == personDTO) {
        throw new BadRequestException("createPOC.person: must not be null");
    } else if (StringUtils.isNotBlank(personDTO.getId())) {
        throw new InvalidInputException("createPOC.person.id: must be null");
    ... ①
```

① business logic is possibly overwhelmed by validation concerns and actual checks

This lecture will introduce working with the Bean Validation API to implement declarative and expressive validation.

Declarative Bean Validation API

```
@Validated(PocValidationGroups.CreatePlusDefault.class)
public PersonPocDTO createPOC(
        @NotNull
        @Valid PersonPocDTO personDTO); ①
```

① conditions well-advertised and isolated from target business logic

1.1. Goals

The student will learn:

- to add declarative pre-conditions and post-conditions to components using the Bean Validation API
- to define declarative validation constraints
- to implement custom validation constraints
- to enable injected call validation for components
- to identify patterns/anti-patterns for validation

1.2. Objectives

At the conclusion of this lecture and related exercises, the student will be able to:

1. add Bean Validation to their project

2. add declarative data validation constraints to types and method parameters
3. configure a `ValidatorFactory` and obtain a `Validator`
4. programmatically validate an object
5. programmatically validate parameters to and response from a method call
6. inspect constraint violations
7. enable Spring/AOP validation for components
8. implement a custom validation constraint
9. implement a cross-parameter validation constraint
10. configure Web API constraint violation responses
11. configure Web API parameter validation
12. configure JPA validation
13. configure Spring Data Mongo Validation
14. identify some patterns/anti-patterns for using validation

Chapter 2. Background

Bean Validation is a standard that originally came out of Java EE/SE as JSR-303 (1.0) in the 2009 timeframe and later updated with JSR-349 (1.1) in 2013, JSR-380 (2.0) in 2017, and finally 3.0 in 2020 with the Java package and XML namespace changes from javax to jakarta.

It was meant to simplify validation — reducing the chance of error and to reduce the clutter of validation within the business code that required validation. The standard is not specific any particular tier (e.g., UI, Web, Service, DB) but has been integrated into several of the individual frameworks. [1]

Implementations include:

- Hibernate Validator
- Apache BVal

Hibernate Validator was the original and current reference implementation and used within Spring Boot today.

[1] *"Jakarta Bean Validation specification"*, Gunnar Morling, 2020

Chapter 3. Dependencies

To get started with validation in Spring Boot — we add a dependency on `spring-boot-starter-validation`.

Validation Dependency

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

That will bring in the validation reference implementation from Hibernate and an implementation for regular expression validation constraints.

Validation Transient Dependencies

```
[INFO] +- org.springframework.boot:spring-boot-starter-validation:jar:3.3.2:compile
[INFO] |  +- org.apache.tomcat.embed:tomcat-embed-el:jar:10.1.26:compile ③
[INFO] |  \- org.hibernate.validator:hibernate-validator:jar:8.0.1.Final:compile ②
[INFO] |     +- jakarta.validation:jakarta.validation-api:jar:3.0.2:compile ①
[INFO] |     +- org.jboss.logging:jboss-logging:jar:3.5.3.Final:compile
[INFO] |     \- com.fasterxml:classmate:jar:1.7.0:compile
```

① overall Bean Validation API

② Bean Validation API reference implementation from Hibernate

③ regular expression implementation for regular expression constraints

Chapter 4. Declarative Constraints

At the core of the Bean Validation API are declarative constraint annotations we can place directly into the source code.

4.1. Data Constraints

The following snippet shows a class with a property that is required to be not-null when valid.

Java Class with Validation Constraint Annotation(s)

```
import jakarta.validation.constraints.NotNull;
...
class AClass {
    @NotNull
    private String aValue;
    ...
```

Constraints do not Actively Prevent Invalid State

The constraint does not actively prevent the property from being set to an invalid value. Unlike with the Lombok annotations, no class code is written as a result of the validation annotations. The constraint will identify whether the property is currently valid when validated. The validating caller can decide what to do with the invalid state.

4.2. Common Built-in Constraints

You can find a list of built-in constraints in the Bean Validation Spec and the Hibernate Validator documentation. A few common ones include:

Table 1. Common Built-in Validator Constraints

• @Null, @NotNull	• @Size(min, max) - value limit
• @NotBlank, @NotEmpty	• @Positive, @Negative
• @Past, @Future	• @PositiveOrZero, @NegativeOrZero
• @Min, Max - collection size	• @Pattern(regex)

Additional constraints are provided by:

- Hibernate Additional Constraints (e.g., @CreditCardNumber)
- Java Bean Validation Extension (JBVExt) (e.g. @Alpha, @IsDate, @IPv4)

We will take a look at how to create a custom constraint later in this lecture.

4.3. Method Constraints

We can provide pre-condition and post-condition constraints on methods and constructors.

The following snippet shows a method that requires a non-null and valid parameter and will return a non-null result. Constraints for input are placed on the individual input parameters. Constraints on the output (as well as cross-parameter constraints) are placed on the method. The `@Validated` annotation is added to components to trigger Spring to enable validation for injected components.

Java Method Declaration with Parameter

```
import jakarta.validation.Valid;
import org.springframework.validation.annotation.Validated;
...
@Component
@Validated ③
public class AService {
    @NotNull ②
    public String aMethod(@NotNull @Valid AClass aParameter) { ①
        return ...;
    }
```

① method requires a non-null parameter with valid content

② the result of the method is required to be non-null

③ `@Validated` triggers Spring's use of the Bean Validation API to validate the call

Null Properties Are Considered @Valid Unless Explicitly Constrained with @NotNull

It is a best practice to consider a null value as valid unless explicitly constrained with `@NotNull`.

We will eventually show all this integrated within Spring, but first we want to make sure we understand the plumbing and what Spring is doing under the covers.

Chapter 5. Programmatic Validation

To work with the Bean Validation API directly, our initial goal is to obtain a standard `jakarta.validation.Validator` instance.

Programmatic Validation Requires Validator

```
import jakarta.validation.Validator;
...
Validator validator;
```

This can be obtained manually or through injection.

5.1. Manual Validator Instantiation

We can get a Validator using one of the `Validation` builder methods to return a `ValidatorFactory`.

The following snippet shows the builder providing an instance of the default factory provider, with the default configuration. If we have no configuration changes, we can simplify with a call to `buildDefaultValidatorFactory()`. The `Validator` instance is obtained from the `ValidatorFactory`. Both the factory and validator instances are thread-safe. We will come back to the `configure()` method options later.

Standard jakarta.validation.Validator Interface

```
import jakarta.validation.Validation;
...
ValidatorFactory myValidatorFactory = Validation.byDefaultProvider()
    .configure()
        //configuration commands
    .buildValidatorFactory(); ①
//ValidatorFactory myValidatorFactory = Validation.buildDefaultValidatorFactory();
Validator myValidator = myValidatorFactory.getValidator(); ①
```

① factory and validator instances are thread-safe, initialized during bean construction, and used during instance methods

5.2. Inject Validator Instance

With the validation starter dependency comes a default Validator. For components, we can simply have it injected.

Injecting Validator

```
@Autowired
private Validator validator;
```

5.3. Customizing Injected Instance

If we want the best of both worlds and have some customizations to make, we can define a `@Bean` factory to replace the AutoConfigure and return our version of the `Validator` instead.

Custom Validator @Bean Factory

```
@Bean ①
public Validator validator() {
    return Validation.byDefaultProvider()
            .configure()
                //configuration commands
            .buildValidatorFactory()
            .getValidator();
}
```

① A custom `Validator` `@Bean` within the application will override the default provided by Spring Boot

5.4. Review: Class with Constraint

The following validation example(s) will use the following class with a non-null constraint on one of its properties.

Example Class with Constraint

```
@Getter
public class AClass {
    @NotNull
    private String aValue;
    ...
```

5.5. Validate Object

The most straight forward use of the validation programmatic API is to validate individual objects. The object to be validated is supplied and a `Set<ConstraintViolation>` is returned. No exceptions are thrown by the Bean Validation API itself for constraint violations. Exceptions are only thrown for invalid use of the API and to report violations within frameworks like Contexts and Dependency Injection (CDI) or Spring Boot.

The following snippet shows an example of using the validator to validate an object with at least one constraint error.

Validate Object

```
//given - @NotNull aProperty set to null by ctor
AClass invalidAClass = new AClass();
//when - checking constraints
```

```
Set<ConstraintViolation<AClass>> violations = myValidator.validate(invalidAClass);①
violations.forEach(v-> log.info("field name={}, value={}, violated={}",
        v.getPropertyPath(), v.getInvalidValue(), v.getMessage()));
//then - there will be at least one violation
then(violations).isNotEmpty(); ②
```

① programmatic call to validate object

② non-empty return set means violations where found

The result of the validation is a `Set<ConstraintViolation>`. Each constraint violation identifies the:

- path to the field in error
- an error message
- invalid value
- descriptors for the annotation and validator

The following shows the output of the example.

Validate Object Text Output

```
field name=aValue, value=null, violated=must not be null
```

Specific Property Validation

We can also validate a value against the definition of a specific property

- `validateProperty(T object, propertyName, groups)`
- `validateValue(Class<T> beanType, propertyName, value, groups)`

5.6. Validate Method Calls

We can also validate calls to and results from methods (and constructors too). This is commonly performed by AOP code — rather than anything closely related to the business logic.

The following snippet shows a class with a method that has input and response constraints. The input parameter must be valid and not null. The response must also be not null. A `@Valid` constraint on an input argument or response will trigger the object validation — which we just demonstrated — to be performed.

Validate Method Calls

```
public class AService {
    @NotNull
    public String aMethod(@NotNull @Valid AClass aParameter) { ① ②
        return ...
    }
```

① `@NotNull` constrains `aParameter` to always be non-null

② `@Valid` triggers validation contents of `aParameter`

With those validation rules in place, we can check them for the following sample call.

Get Reference to Target Service and Input Parameters

```
//given
AService myService = new AService(); ①
AClass myValue = new AClass();
//when
String result = myService.aMethod(myValue);
```

① Note: Service shown here as POJO. Must be injected for container to intercept and subject to validation

Please note that the code above is a plain POJO call. Validation is only automatically performed for injected components. We will use this call to describe how to programmatically validate a method call.

5.7. Identify Method Using Java Reflection

Before we can validate anything, we must identify the descriptors of the call and resolve a `Method` reference using Java Reflection.

In the following example snippet we locate the method called `aMethod` on the `AService` class that accepts one parameter of `AClass` type.

Identify Method to Call using Java Reflection

```
Object[] methodParams = new Object[]{ myValue };
Class<?>[] methodParamTypes = new Class<?>[]{ AClass.class };
Method methodToCall = AService.class.getMethod("aMethod", methodParamTypes);
```

The code above has now resolved a reference to the following method call.

Resolved Method Reference

```
(AService)myService).aMethod((AClass)myValue);
```

5.8. Programmatically Check for Parameter Violations

Without actually making the call, we can check whether the given parameters violate defined method constraints by accessing the `ExecutableValidator` from the `Validator` object. `Executable` is a generalized `java.lang.reflect` type for `Method` and `Constructor`.

Programmatically Check For Parameter Violations

```
//when
Set<ConstraintViolation<AService>> violations = validator
    .forExecutables() ①
```

```
    .validateParameters(myService, methodToCall, methodParams);
```

① returns `ExecutableValidator`

The following snippet shows the reporting of the validation results when subjecting our `myValue` parameter to the defined validation rules of the `aMethod()` method.

Invalid Input is Identified

```
//then
then(violations).hasSize(1);
ConstraintViolation<?> violation = violations.iterator().next();
then(violation.getPropertyPath().toString()).isEqualTo("aMethod.arg0.aValue");
then(violation.getMessage()).isEqualTo("must not be null");
then(violation.getInvalidValue()).isEqualTo(null);
then(violation.getInvalidValue()).isEqualTo(myValue.getAValue());
```

5.9. Validate Method Results

We can also validate what is returned against the defined rules of the `aMethod()` method using the same service instance and method reflection references from the parameter validation. Except in this case, `methodToCall` has already been called, and we are now holding onto the result value.

The following example shows an example of validating a null result against the return rules of the `aMethod()` method.

Validating Value Relative to Method Return Value Constraints

```
//given
String nullResult = null;
//when
violations = validator.forExecutables()
        .validateReturnValue(myService, methodToCall, nullResult);
```

Since null is not allowed, one violation is reported.

Method Return Value Constraint Violation(s)

```
//then
then(violations).hasSize(1);
violation = violations.iterator().next();
then(violation.getPropertyPath().toString()).isEqualTo("aMethod.<return value>");
then(violation.getMessage()).isEqualTo("must not be null");
then(violation.getInvalidValue()).isEqualTo(nullResult);
```

Chapter 6. Method Parameter Naming

Validation is able to easily gather meaningful field path information from classes and properties. When we validated the `AClass` instance, we were told the given name of the property in error supplied from reflection.

Java Class with Property

```
class AClass {
    @NotNull
    private String aValue;
...
```

Validation Result using Property Name

```
field name=aValue, value=null, violated=must not be null
```

However, reflection by default does not provide the given names of parameters — only the position.

Java Method Declaration with Parameter

```
public class AService {
    @NotNull
    public String aMethod(@NotNull @Valid AClass aParameter) {
        return ...
    }
```

Validation result using Argument Position

```
[ERROR]   SelfDeclaredValidatorTest.method_arguments:96
expected: "aMethod.aParameter.aValue"
 but was: "aMethod.arg0.aValue" ①
```

① By default, argument position supplied (`arg0`) — not argument name

There are two ways to solve this.

6.1. Option 1: Add -parameters to Java Compiler Command

The first way to solve this would be to add the `-parameters` option to the Java compiler.

The following snippet shows how to do this for the `maven-compiler-plugin`. Note that this only applies to what is compiled with Maven and not what is actively worked on within the IDE.

Add -parameters to Maven Compile Command

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
        <parameters>true</parameters>
    </configuration>
</plugin>
```

The above appears to work fine with the Maven compiler plugin 3.10.1, but I encountered issues getting that working with the older 3.8.1 (without an explicit `-parameters` in `compilerArgs`).

6.2. Option 2: Add Custom ParameterNameProvider

Another way to help provide parameter names is to configure the `ValidatorFactory` with a `ParameterNameProvider`.

Add Custom ParameterNameProvider

```
ValidatorFactory myValidatorFactory = Validation.byDefaultProvider()
    .configure()
        .parameterNameProvider(new MyParameterNameProvider()) ①
    .buildValidatorFactory();
```

① configuring `ValidatorFactory` with custom parameter name provider

6.2.1. ParameterNameProvider

The following snippets show the skeletal structure of a sample `ParameterNameProvider`. It has separate incoming calls for `Method` and `Constructor` calls and must produce a list of names to use. This particular example is simply returning the default. Example work will be supplied next.

Custom Parameter Name Provider Skeletal Shell

```
import jakarta.validation.ParameterNameProvider;
import java.lang.reflect.Constructor;
import java.lang.reflect.Executable;
import java.lang.reflect.Method;
import java.lang.reflect.Parameter;
...
public class MyParameterNameProvider implements ParameterNameProvider {
    @Override
    public List<String> getParameterNames(Constructor<?> ctor) {
        return getParameterNames((Executable) ctor);
    }
    @Override
    public List<String> getParameterNames(Method method) {
```

```
        return getParameterNames((Executable) method);
    }
    protected List<String> getParameterNames(Executable method) {
        List<String> argNames = new ArrayList<>(method.getParameterCount());
        for (Parameter p: method.getParameters()) {
            //do something to determine Parameter p's desired name
            String argName=p.getName(); ①
            argNames.add(argName);
        }
        return argNames; ②
    }
}
```

① real work to determine the parameter name goes here

② must return a list of parameter names of expected size

6.2.2. Named Parameters

The Bean Validation API does not provide a way to annotate parameters with names. They left that up to us and other Java standards. In this example, I am making use of `jakarta.inject.Named` to supply a textual name of my choice.

Java Method Declaration with @Named Parameter

```
import jakarta.inject.Named;
...
private static class AService {
    @NotNull
    public String aMethod(@NotNull @Valid @Named("aParameter") AClass aParameter) {①
        return ...
    }
```

① `@Named` annotation is providing a name to use for `MyParameterNameProvider`

6.2.3. Determining Parameter Name

Now we can update `MyParameterNameProvider` to look for and use the `@Named.value` property if provided or default to the name from reflection.

Processing @Named Annotation to Obtain Parameter Name

```
Named named = p.getAnnotation(Named.class);
String argName=named!=null && StringUtils.isNotBlank(named.value()) ?
        argName=named.value() : //@Named.property
        p.getName();            //default reflection name
argNames.add(argName);
```

The result is a property path that possibly has more meaning.

Before/After Parameter Naming Solution(s) Applied

```
original: aMethod.arg0.aValue
assisted: aMethod.aParameter.aValue
```

Chapter 7. Graphs

Constraint validation can follow a graph of references annotated with `@Valid`.

The following snippet shows an example set of parent classes — each with a reference to equivalent child instances. The child instance will be invalid in both cases. Only the `TraversingParent` will be considered invalid because only that parent class defines a requirement that the child be valid (using `@Valid`).

Validation Graph Example Classes

```
class Child {
    @NotNull
    String cannotBeNull; ①
}
class NonTraversingParent {
    Child child = new Child(); ②
}
class TraversingParent {
    @Valid ④
    Child child = new Child(); ③
}
```

① child attribute constrained to not be null

② child instantiated with default instance but not annotated

③ child instantiated with default instance

④ annotation instructs validator to traverse to the child and validate

7.1. Graph Non-Traversal

We know from the previous chapter that we can validate any constraints on an object by passing the instance to the `validate()` method. However, validation will stop there if there are no `@Valid` annotations on references.

The following snippet shows an example of a parent with an invalid child, but due to the lack of `@Valid` annotation, the child state is not evaluated with the parent.

No Validation Traversal to Child

```
//given
Object nonTraversing = new NonTraversingParent(); ①
//when
Set<ConstraintViolation<Object>> violations = validator.validate(nonTraversing); ②
//then
then(violations).isEmpty(); ③
```

① parent contains an invalid child

② constraint validation does not traverse from parent to child

③ child errors are not reported because they were never checked

7.2. Graph Traversal

Adding the `@Valid` annotation to an object reference activates traversal to and validation of the child instance. This can be continued to grandchildren with follow-on child `@Valid` annotations.

The following snippet shows an example of a parent whose validation traverses to the child because of the `@Valid` annotation.

Validation Traversal to Child

```
import jakarta.validation.Valid;
...
//given
Object traversing = new TraversingParent(); ①
//when
Set<ConstraintViolation<Object>> violations = validator.validate(traversing); ②
//then
String errorMsgs = violations.stream()
        .map(v->v.getPropertyPath().toString()+":"+v.getMessage())
        .collect(Collectors.joining("\n"));
then(errorMsgs).contains("child.cannotBeNull:must not be null"); ③
then(violations).hasSize(1);
```

① parent contains an invalid child

② constraint validation traverses relationship and performed on parent and child

③ child errors reported

Chapter 8. Groups

The Bean Validation API supports validation within different contexts using groups. This allows us to write constraints for specific situations, use them when appropriate, and bypass them when not pertinent. The earlier examples all used the default `jakarta.validation.groups.Default` group and were evaluated by default because no group was specified in the call to `validate()`.

We can define our own custom groups using Java interfaces.

8.1. Custom Validation Groups

The following snippet shows an example of two groups. `Create` should only be applied during creation. `CreatePlusDefault` should only be applied during creation but will also apply default validation. `UpdatePlusDefault` can be used to denote constraints unique to updates.

Custom Validation Groups

```
import jakarta.validation.groups.Default;
...
public interface PocValidationGroups { ③
    interface Create{} ①
    interface CreatePlusDefault extends Create, Default{} ②
    interface UpdatePlusDefault extends Default{}
```

① custom group name to be used during create

② groups that extend another group have constraints for that group applied as well

③ outer interface is not required, Used in example to make the purpose and source of the group obvious

Each interface is used in two ways:

1. **on the constraint** to denote the group it applies to. Any constraint annotated with a group (`Create`) will get validated if validation is triggered for that same type (`Create`) or subtype (`CreatePlusDefault`).
2. denote the **constraint types to validate**. The constraint types validated will be of the same type or a sub-type.

8.2. Applying Groups

We can assign specific groups to a constraint. In the following example,

- `@Null id` will only be validated when validating the `Create` or `CreatePlusDefault` groups
- `@Past dob` will be validated for both `CreatePlusDefault` and `Default` validation
- `@Size contactPoints` and `@NotNull contactPoints` will each be validated the same as `@Past dob`. The default group is `Default` when left unspecified.

```
public class PersonPocDTO {
    @Null(groups = PocValidationGroups.Create.class, ①
        message = "cannot be specified for create")
    private String id;
    private String firstName;
    private String lastName;
    @Past(groups = Default.class) ②
    private LocalDate dob;
    @Size(min=1, message = "must have at least one contact point") ③
    private List<@NotNull @Valid ContactPointDTO> contactPoints;
```

① explicitly setting group to `Create`, which does not include `Default`

② explicitly setting group to `Default`

③ implicitly setting group to `Default`

8.3. Skipping Groups

With use-case-specific groups assigned, we can have certain defined constraints ignored.

The following example shows the validation of an object. It has an assigned `id`, which would make it invalid for a create. However, there are no violations reported because the group for the `@Null id` constraint was not validated because its assigned group (`Create`) is not a sub-type of `Default`.

Validation Group Skipped Example

```
//given
ContactPointDTO invalidForCreate = contactDTOFactory.make(ContactDTOFactory.oneUpId);
①
//when
Set<ConstraintViolation<ContactPointDTO>> violations = validator.validate
(invalidForCreate); ②
//then
then(violations).hasSize(0);
```

① object contains non-null `id`, which is invalid for create scenarios

② implicitly validating against the default group. `Create` group constraints not validated

Can Redefine Default Group for Type with @GroupSequence

The Bean Validation API makes it possible to redefined the default group for a particular type using a `@GroupSequence`.

8.4. Applying Groups

To apply a non-default group to the validation — we can simply add their interface identifiers in a sequence after the object passed to `validate()`.

The following snippet shows an example of the `CreatePlusDefault` group being applied. The `@Null id`

constraint is validated and reported in error because the group it was assigned to was part of the validation command.

Validation Group Applied Example

```
...
//when
violations = validator.validate(invalidForCreate, CreatePlusDefault.class); ①
//then
then(violations).hasSize(1); ②
then(errors(violations)).contains("id:cannot be specified for create"); ③
```

① validating both the `CreatePlusDefault` and `Default` groups

② `@Null id` violation detected and reported

③ `errors()` is a local helper method written to extract field and text from violation

Chapter 9. Multiple Groups

We have two ways of treating multiple groups

validate all

performed by passing more than one group to the `validate()` method. Each group is validated in a non-deterministic manner

short circuit

performed by defining a `@GroupSequence`. Each group is validated in order, and the sequence is short-circuited when there is a failure.

9.1. Example Class with Different Groups

The following snippet shows an example of a class with validations that perform at different costs.

- `@Size email` is thought to be straightforward to validate
- `@Email email` is thought to be a more detailed validation
- the remaining validations have not been included in this class hierarchy

Class with Different Groups

```
public class ContactPointDTO {
    @Null (groups = {Create.class},
            message = "cannot be specified for create")
    private String id;
    @NotNull
    private String name;
    @Size(min=7, max=40, groups= SimplePlusDefault.class) ①
    @Email(groups = DetailedOnly.class) ②
    private String email;
```

① `@Size email` is thought to be a cheap sanity check, but overly simplistic

② `@Email email` is thought to be thorough validation, but only worth it for reasonably sane values

The following snippet shows the groups being used in this example.

Example Groups

```
interface Create{}
interface SimplePlusDefault extends Default {}
interface DetailedOnly {}
```

9.2. Validate All Supplied Groups

When groups are passed to validate in a sequence, all groups in that sequence are validated.

The following snippet shows an example with `SimplePlusDefault` and `DetailedOnly` supplied to `validate()`. Each group will be validated, no matter what the results are.

Validate All Supplied Groups Example

```
String nameErr="name:must not be null"; //Default Group
String sizeErr="email:size must be between 7 and 40"; //Simple Group
String formatErr="email:must be a well-formed email address"; //DetailedOnly Group
//when - validating against all groups
Set<ConstraintViolation<ContactPointDTO>> violations = validator.validate(
        invalidContact,
        SimplePlusDefault.class, DetailedOnly.class);
//then - all groups will have their violations reported
then(errors(violations)).contains(nameErr, sizeErr, formatErr).hasSize(3); ① ② ③
```

① `@NotNull name` (`nameError`) is part of `Default` group

② `@Size email` (`sizeError`) is part of `SimplePlusDefault` group

③ `@Email email` (`formatError`) is part of `DetailedOnly` group

9.3. Short-Circuit Validation

If we instead want to layer validations such that cheap validations come first and more extensive or expensive validations occur only after earlier groups are successful, we can define a `@GroupSequence`.

- Groups earlier in the sequence are performed first.
- Groups later in the sequence are performed later — but only if all constraints in earlier groups pass. Validation will short-circuit at the individual group level when applying a sequence.

The following snippet shows an example of defining a `@GroupSequence` that lists the order of group validation.

Example @GroupSequence

```
@GroupSequence({ SimplePlusDefault.class, DetailedOnly.class }) ①
public interface DetailOrder {};
```

① defines an order-list of validation groups to apply

The following example shows how the validation stopped at the `SimplePlusDefault` group and did not advance to the `DetailedOnly` group.

@GroupSequence Use Example

```
//when - validating using a @GroupSequence
violations = validator.validate(invalidContact, DetailOrder.class);
//then - validation stops once a group produces a violation
then(errors(violations)).contains(nameErr, sizeErr).hasSize(2); ①
```

① validation was short-circuited at the group where the first set of errors detected

9.4. Override Default Group

The `@GroupSequence` annotation can be directly applied to a type to override the default group when validating instances of that class.

Chapter 10. Spring Integration

We saw earlier how we could programmatically validate constraints for Java methods. This capability was not intended for business code to call — but rather for calls to be intercepted by AOP and constraints applied by that intercepting code. We can annotate `@Component` classes or interfaces with constraints and have Spring perform that validation role for us.

The following snippet shows an example of an interface with a simple `aCall` method that accepts an `int` parameter that must be greater than 5. All the information on the method call should be familiar to you by now. Only the `@Validated` annotation is new. The `@Validated` annotation triggers Spring AOP to apply Bean Validation to calls made on the type (interface or class).

Component Interface

```
import org.springframework.validation.annotation.Validated;
import jakarta.validation.constraints.Min;

@Validated ②
public interface ValidatedComponent {
    void aCall(@Min(5) int mustBeGE5); ①
}
```

① interface defines constraint for parameter(s)

② `@Validated` triggers Spring to perform method validation on component calls

10.1. Validated Component

The following snippet shows a class implementation of the interface and further declared as a `@Component`. Therefore, it can be injected and method calls subject to container interpose using AOP interception.

Component Implementation

```
@Component ①
public class ValidatedComponentImpl implements ValidatedComponent {
   @Override
   public void aCall(int mustBeGE5) {
   }
}
```

① designates this class implementation to be used for injection

The component is injected into clients.

Component Injection

```
@Autowired
private ValidatedComponent component;
```

10.2. ConstraintViolationException

With the component injected, we can have parameters and results validated against constraints.

The following snippet shows an example component call where a call is made with an invalid parameter. Spring performs the method validation, throws a `jakarta.validation.ConstraintViolationException`, and prevents the call. The `Set<ConstraintViolation>` can be obtained from the exception. At that point, we return to familiar territory we covered with programmatic validation.

Injected Component Validation by Spring

```
import jakarta.validation.ConstraintViolationException;
...
//when
ConstraintViolationException ex = catchThrowableOfType(
        () -> component.aCall(1), ①
        ConstraintViolationException.class);
//then
Set<ConstraintViolation<?>> violations = ex.getConstraintViolations();
String errorMsgs = violations.stream()
        .map(v->v.getPropertyPath().toString() +":" + v.getMessage())
        .collect(Collectors.joining("\n"));
then(errorMsgs).isEqualTo("aCall.mustBeGE5:must be greater than or equal to 5");
```

① Spring intercepts the component call, detects violations, and reports using exception

10.3. Successful Validation

Of course, if we pass valid parameter(s) to the method —

- the parameters are validated against the method constraints
- no exception is thrown
- the `@Component` method is invoked
- the return object is validated against declared constraints (none in this example)

Example Successful Call

```
assertThatNoException().isThrownBy(
        ()->component.aCall(10) ①
    );
```

① parameter value `10` satisfies the `@Min(5)` constraint — thus no exception

10.4. Liskov Substitution Principle

One thing you may have noticed with the selected example is that the interface contained constraints and not the class declaration. As a matter of fact, if we add any additional constraint

beyond what the interface defined — we will get a `ConstraintDeclarationException` thrown — preventing the call from completing. The Bean Validation Specification describes it as following the Liskov Substitution Principle — where anything that is a subtype of `T` can be inserted in place of `T`. Said more specific to Bean Validation — a subtype or implementation class method cannot add more restrictive constraints to call.

```
@Validated
public interface ValidatedComponent {
    void aCall(@Min(5) int mustBeGE5);
}

@Component
public class ValidatedComponentImpl implements ValidatedComponent {
@Override
    public void aCall(@Positive int mustBeGE5) {} //invalid ①
}
```

① Bean Validation enforces that subtypes cannot be more constraining than their interface or parent type(s)

Liskov Violation Error Message Example

```
jakarta.validation.ConstraintDeclarationException: HV000151: A method overriding
another method must not redefine the parameter constraint configuration, but method
ValidatedComponentImpl#aCall(int) redefines the configuration of
ValidatedComponent#aCall(int).
```

10.5. Disabling Parameter Constraint Override

For the Hibernate Validator, the constraint override rule can be turned off during factory configuration. You can find other Hibernate-specific features in the Hibernate Validator Specifics section of the on-line documentation.

The snippet below uses a generic property interface to disable parameter override constraint.

Generic Property Setting

```
return Validation.byDefaultProvider() ①
    .configure()
        .addProperty("hibernate.validator.allow_parameter_constraint_override",
                                                Boolean.TRUE.toString()) ②
        .parameterNameProvider(new MyParameterNameProvider())
    .buildValidatorFactory()
    .getValidator();
```

① generic factory configuration interface used to initialize factory

② generic property interface used to set custom behavior of Hibernate Validator

The snippet below uses a Hibernate-specific configurer and custom method to disable parameter override constraint.

Hibernate Specific API Supported Property Setting

```
return Validation.byProvider(HibernateValidator.class) ①
    .configure()
        .allowOverridingMethodAlterParameterConstraint(true) ②
        .parameterNameProvider(new MyParameterNameProvider())
    .buildValidatorFactory()
    .getValidator();
```

① Hibernate-specific configuration interface used to initialize factory

② Hibernate-specific method used to set custom behavior of Hibernate Validator

10.6. Spring Validated Group(s)

We saw earlier how we could programmatically validate using explicit validation groups. Spring uses the `@Validated` annotation in a dual role to define that as well.

- `@Validated` on the interface/class triggers validation to occur
- `@Validated` on a parameter or method causes the validation to apply the identified group(s)
  - the `groups` attribute is used for this purpose

Declarative Validation Group Assignment for Method

```
//@Validated ①
public interface PocService {
    @NotNull
    @Validated(CreatePlusDefault.class) ②
    PersonPocDTO createPOC(
            @NotNull ③
            @Valid PersonPocDTO personDTO); ④
```

① `@Validated` at the class/interface/component level triggers validation to be performed

② `@Validated` at the method level used to apply specific validation groups (`CreatePlusDefault`)

③ `@NotNull` at the property level requires `personDTO` to be supplied

④ `@Valid` at the property level triggered `personDTO` to be validated

10.7. Spring Validated Group(s) Example

The following snippet shows an example of a class where the `id` property is required to be null when validating against the `Create` group.

PersonPocDTO id Property

```
public class PersonPocDTO {
```

```
    @Null(groups = Create.class, message = "cannot be specified for create")
    private String id; ①
```

① `id` must be null only when validating against `Create` group

The following snippet shows the constrained method being passed a parameter that is illegal for the `Create` constraint group. A `ConstraintViolationException` is thrown with violations.

Spring Group Validation Example

```
PersonPocDTO pocWithId = pocFactory.make(oneUpId); ③
assertThatThrownBy(() -> pocService.createPOC(pocWithId)) ①
        .isInstanceOf(ConstraintViolationException.class)
        .hasMessageContaining("createPOC.person.id: cannot be specified for create");
②
```

① `@Validated` on component triggered validation to occur

② `@Validated(CreatePlusDefault.class)` caused `Create` and `Default` rules to be validated

③ poc instance created with an `id` assigned — making it invalid

Chapter 11. Custom Validation

Earlier I listed several common, built-in constraints and available library constraints. Hopefully, they provide most or all of what is necessary to meet our validation needs — but there is always going to be that need for custom validation.

The snippet below shows an example of a custom validation being applied to a `LocalDate` — that validates the value is of a certain age in years, with an optional timezone offset.

@MinAge Custom Constraint Example Usage

```
public class ValidatedClass {
    @MinAge(age = 16, tzOffsetHours = -4)
    private LocalDate dob;
```

11.1. Constraint Interface Definition

We can start with the interface definition for our custom constraint annotation.

Example Validation Constraint Interface

```
@Documented
@Target({ ElementType.METHOD, FIELD, ANNOTATION_TYPE, PARAMETER, TYPE_USE })
@Retention( RetentionPolicy.RUNTIME )
@Repeatable(value= MinAge.List.class)
@Constraint(validatedBy = {
        MinAgeLocalDateValidator.class,
        MinAgeDateValidator.class
})
public @interface MinAge {
    String message() default "age below minimum({age}) age";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};

    int age() default 0;
    int tzOffsetHours() default 0;

    @Documented
    @Retention(RUNTIME)
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, PARAMETER, TYPE_PARAMETER })
    @interface List {
        MinAge[] value();
    }
}
```

11.2. @Documented Annotation

The `@Documented` annotation instructs the Javadoc processing to include the Javadoc for this annotation within the Javadoc output for the classes that use it.

@Documented Annotation

```
/**
 * Defines a minimum age based upon a LocalDate, the current
 * LocalDate, and a specified timezone.
 */

@Documented //include this in Javadoc for elements that it is defined
```

The following images show the impact made to Javadoc for a different `@PersonHasName` annotation example. Not only are the constraints shown for the class, but the documentation for the annotations is included in the produced Javadoc.

Figure 1. PersonPocDTO Javadoc

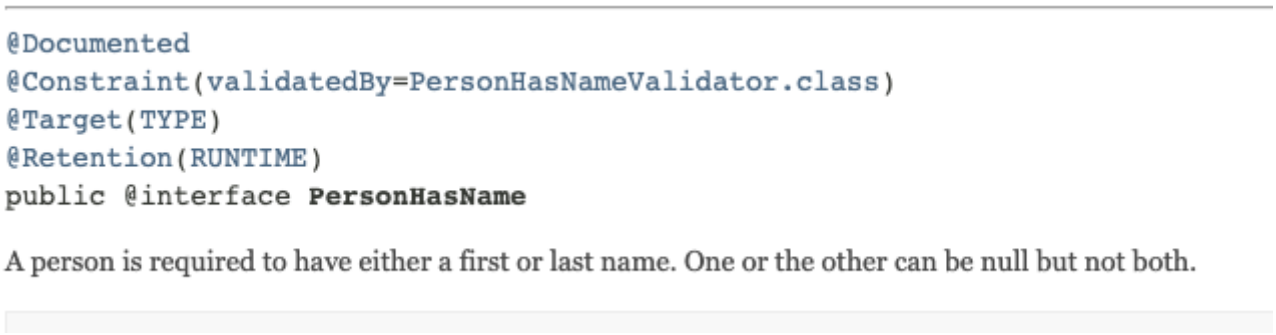


Figure 2. @PersonHasName Annotation Javadoc

11.3. @Target Annotation

The `@Target` annotation defines locations where the constraint is legally allowed to be applied. The following table lists examples of the different target types.

Table 2. Annotation @Target ElementTypes

ElementType.FIELD

```
@MinAge LocalDate dob;
```

define validation on a Java attribute within a class

ElementType.METHOD

```
@MinAge LocalDate getDob(); ①
@MinAge void add(LocalDate dob,
LocalDate dateOfHire);②
```

① `@MinAge` being used as return value constraint here

② `@MinAge` being used as cross-param constraint here

define validation on a return value or cross-parameters of a method

ElementType.PARAMETER

```
void method(@MinAge LocalDate dob){}
```

define validation on a parameter to a method

ElementType.TYPE_USE

```
List<@MinAge LocalDate> dobs;
```

define validation within a parameterized type

ElementType.TYPE

```
@MinAge
class Person {
    LocalDate dob;
}
```

define validation on an interface or class that likely inspects the state of the type

ElementType.CONSTRUCTOR

```
class Person {
    LocalDate dob;
    @MinAge Person() {}
}
```

define validation on the resulting instance after constructor completes

ElementType.ANNOTATION_TYPE

```
public @interface MinAge {}

@MinAge(age=18)
public @interface AdultAge {...
```

This type allows other annotations to be defined based on this annotation. The snippet shows an example of constraint `@AdultAge` to be implemented as `@MinAge(age=18)`

11.4. @Retention

`@Retention` is used to determine the lifetime of the annotation.

Annotation @Retention

```
@Retention(
```

```
    //SOURCE - annotation discarded by compiler
    //CLASS - annotation available in a class file but not loaded at runtime - default
    RetentionPolicy.RUNTIME //annotation available through reflection at runtime
)
```

Bean Validation should always use `RUNTIME`

11.5. @Repeatable

The `@Repeatable` annotation and declaration of an annotation wrapper class is required to supply annotations multiple times on the same target. This is normally used in conjunction with different validation groups. The `@Repeatable.value` specifies an @interface that contains a `value` method that returns an array of the annotation type.

The snippet below provides an example of the `@Repeatable` portions of `MinAge`.

Enabling @Repeatable

```
@Repeatable(value= MinAge.List.class)
public @interface MinAge {
...
    @Retention(RUNTIME)
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, PARAMETER, TYPE_USE })
    @interface List {
        MinAge[] value();
    }
}
```

The following snippet shows the annotation being applied multiple times to the same property — but assigned different groups.

Example @Repeatable Use

```
@MinAge(age=18, groups = {VotingGroup.class})
@MinAge(age=65, groups = {RetiringGroup.class})
public LocalDate getConditionalDOB() {
    return dob;
}
```

Repeatable Syntax Use Simplified

The requirement for the wrapper class is based on the Java requirement to have only one annotation type per target. Prior to Java 8, we were also required to explicitly use the construct in the code. Now it is applied behind the scenes by the compiler.

Pre-Java 8 Use of Repeatable

```
@MinAge.List({
```

```
  @MinAge(age=18, groups = {VotingGroup.class})
  @MinAge(age=65, groups = {RetiringGroup.class})
})
public LocalDate getConditionalDOB() {
```

11.6. @Constraint

The `@Constraint` is used to identify the class(es) that will implement the constraint. The annotation is not used for constraints built upon other constraints (e.g., `@AdultAge` ⇒ `@MinAge`). The annotation can specify multiple classes — one for each unique type the constraint can be applied to.

The following snippet shows two validation classes: one for `java.util.Date` and the other for `java.time.LocalDate`.

@Constraint

```
@Constraint(validatedBy = {
        MinAgeLocalDateValidator.class, ①
        MinAgeDateValidator.class ②
})
public @interface MinAge {
```

① validates annotated `LocalDate` values

② validates annotated `Date` values

Constraining Different Types

```
@MinAge(age=18, groups = {VotingGroup.class})
@MinAge(age=65, groups = {RetiringGroup.class})
public LocalDate getConditionalDOB() { ①
    return dob;
}

@MinAge(age=16, message="found java.util.Date age({age}) violation")
public Date getDobAsDate() { ②
    return Date.from(dob.atStartOfDay().toInstant(ZoneOffset.UTC));
}
```

① constraining type `LocalDate`

② constraining type `Date`

11.6.1. Core Constraint Annotation Properties

The core constraint annotation properties include

message

contains the default error message template to be returned when constraint violated. The contents of the message get interpolated to fill in variables and substitute entire text strings. This

provides a means for more detailed messages as well as internationalization of messages.

groups

identifies which group(s) to validate this constraint against

payload

used to supply instance-specific metadata to the validator. A common example is to establish a severity structure to instruct the validator how to react.

The following snippet provides an example declaration of core properties for `@MinAge` constraint.

Core Constraint Annotation Properties

```
public @interface MinAge {
    String message() default "age below minimum({age}) age";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
    ...
```

11.7. @MinAge-specific Properties

Each constraint annotation can also define its own unique properties. These values will be expressed in the target code and made available to the constraining code at runtime.

The following example shows the `@MinAge` constraint with two additional properties

- age - defines how old the subject has to be in years to be valid
- tzOffsetHours - an example property demonstrating we can have as many as we need

@MinAge-specific Properties

```
public @interface MinAge {
...
    int age() default 0;
    int tzOffsetHours() default 0;
...
```

11.8. Constraint Implementation Class

The annotation referenced zero or more constraint implementation classes — differentiated by the Java type they can process.

@Constraint

```
@Constraint(validatedBy = {
        MinAgeLocalDateValidator.class,
        MinAgeDateValidator.class
})
```

```
public @interface MinAge {
```

Each implementation class has two methods they can override.

- `initialize()` accepts the specific annotation instance that will be validated against
- `isValid()` accepts the value to be validated and a context for this specific call. The minimal job of this method is to return true or false. It can optionally provide additional or custom details using the context.

11.9. Constraint Implementation Type Examples

The following snippets show the `@MinAge` constraint being implemented against two different temporal types: `java.time.LocalDate` and `java.util.Date`. We, of course, could have used inheritance to simplify the implementation.

@MinAge java.time.LocalDate Constraint Implementation Class

```
public class MinAgeLocalDateValidator implements ConstraintValidator<MinAge,
LocalDate> {
    ...
    @Override
    public void initialize(MinAge annotation) { ... }
    @Override
    public boolean isValid(LocalDate dob, ConstraintValidatorContext context) { ... }
```

@MinAge java.util.Date Constraint Implementation Class

```
public class MinAgeDateValidator implements ConstraintValidator<MinAge, Date> {
    ...
    @Override
    public void initialize(MinAge annotation) { ... }
    @Override
    public boolean isValid(Date dob, ConstraintValidatorContext context) { ... }
```

11.10. Constraint Initialization

The constraint initialize provides a chance to validate whether the constraint definition is valid on its own. An invalid constraint definition is reported using a `RuntimeException`. If an exception is thrown during either the `initialize()` or `isValid()` method, it will be wrapped in a `ValidationException` before being reported to the application.

Constraint Initialization

```
public class MinAgeLocalDateValidator implements ConstraintValidator<MinAge,
LocalDate> {
  private int minAge;
  private ZoneOffset zoneOffset;
```

```
  @Override
  public void initialize(MinAge annotation) {
    if (annotation.age() < 0) {
        throw new IllegalArgumentException("age constraint cannot be negative");
    }
    this.minAge = annotation.age();

    if (annotation.tzOffsetHours() > 23 || annotation.tzOffsetHours() < -23) {
        throw new IllegalArgumentException("tzOffsetHours must be between -23 and +23
");
    }
    zoneOffset = ZoneOffset.ofHours(annotation.tzOffsetHours());
  }
```

11.11. Constraint Validation

The `isValid()` method is required to return a boolean true or false — to indicate whether the value is valid, according to the constraint. It is a best-practice to only validate non-null values and to independently use `@NotNull` to enforce a required value.

The following snippet shows a simple evaluation of whether the expressed `LocalDate` value is older than the minimum required `age`.

Constraint Validation

```
    @Override
    public boolean isValid(LocalDate dob, ConstraintValidatorContext context) {
        if (null==dob) { //assume null is valid and use @NotNull if it should not be
            return true;
        }
        final LocalDate now = LocalDate.now(zoneOffset);
        final int currentAge = Period.between(dob, now).getYears();
        return currentAge >= minAge;
    }
}
```

Treat Null Values as Valid

Null values should be considered valid and independently constrained by `@NotNull`.

11.12. Custom Violation Messages

I won't go into any detail here, but will point out that the `isValid()` method has the opportunity to either augment or replace the constraint violation messages reported.

The following example is from a cross-parameter constraint and is reporting that parameters 1 and 2 are not valid when used together in a method call.

Custom Violation Messages

```
context.buildConstraintViolationWithTemplate(context.getDefaultConstraintMessageTempla
te())
        .addParameterNode(1)
        .addConstraintViolation()
        .buildConstraintViolationWithTemplate(context
.getDefaultConstraintMessageTemplate())
        .addParameterNode(2)
        .addConstraintViolation();
//the following removes default-generated message
//context.disableDefaultConstraintViolation(); ①
```

① make this call to eliminate the default message

The following shows the default constraint message provided in the target code.

Default Constraint Message Provided

```
@ConsistentNameParameters(message = "name1 and/or name2 must be supplied") ①
public NamedThing(String id, String name1, String name2, LocalDate dob) {
```

① `@ConsistentNameParameters` is a cross-parameter validation constraint validating name1 and name2

Generated Violation Message Paths

```
NamedThing.name1:name1 and/or name2 must be supplied ①
NamedThing.name2:name1 and/or name2 must be supplied ①
NamedThing.<cross-parameter>:name1 and/or name2 must be supplied ②
```

① path/message generated by the custom constraint validator

② default path/message generated by validation framework

Chapter 12. Cross-Parameter Validation

Custom validation is useful, but often times the customization is necessary for when we need to validate two or more parameters used together.

The following snippet shows an example of two parameters — name1 and name2 — with the requirement that at least one be supplied. One or the other can be null — but not both.

Cross-Parameter Constraint Use Example

```
class NamedThing {
    @ConsistentNameParameters(message = "name1 and/or name2 must be supplied") ①
    public NamedThing(String id, String name1, String name2, LocalDate dob) {
```

① cross-parameter annotation placed on the method

12.1. Cross-Parameter Annotation

The cross-parameter constraint will likely only apply to a method or constructor, so the number of `@Targets` will be more limited. Other than that — the differences are not yet clear that it is performing special validation. Something else will be necessary in the `ConstraintValidator` class.

Cross-Parameter Annotation

```
@Documented
@Constraint(validatedBy = ConsistentNameParameters.ConsistentNameParametersValidator
.class )
@Target({ElementType.METHOD, ElementType.CONSTRUCTOR})
@Retention(RetentionPolicy.RUNTIME)
public @interface ConsistentNameParameters {
```

12.2. @SupportedValidationTarget

Because of the ambiguity when annotating a method, we need to apply the `@SupportedValidationTarget` annotation to identify whether the validation is for the parameters going into the method or the response from the method.

- ValidationTarget.PARAMETERS - parameters to method
- ValidationTarget.ANNOTATED_ELEMENT - returned element from method

Example Cross-Parameter Validator Declaration

```
@SupportedValidationTarget(ValidationTarget.PARAMETERS) ①
public class ConsistentNameParametersValidator
        implements ConstraintValidator<ConsistentNameParameters, Object[]> { ②
```

① declaring that we are validating parameters going into method/ctor

② must accept `Object[]` that will be populated with actual parameters

@SupportValidationTarget adds Clarity to Annotation Purpose

Think how the framework would be confused without the `@SupportedValidationTarget` annotation if we wanted to validate a method that returned an `Object[]`. The framework would not know whether to pass us the parameters or the response object.

12.3. Method Call Correctness Validation

Funny - within the work of a validation method, it sometimes needs to validate whether it is being called correctly. Was the constraint annotation applied to a method with the wrong signature? Did — somehow — a parameter of the wrong type end up in an unexpected position?

The snippet below highlights the point that cross-parameter constraint validators are strongly tied to method signatures. They expect the parameters to be validated in a specific position in the array and to be of a specific type.

Validating Correct Method Call

```
@Override
public boolean isValid(Object[] values, ConstraintValidatorContext context) { ①
    if (values.length != 4) { ②
        throw new IllegalArgumentException(
            String.format("Unexpected method signature, 4 params expected, %d
supplied", values.length));
    }
    for (int i=1; i<3; i++) { //look at positions 1 and 2 ③
        if (values[i]!=null && !(values[i] instanceof String)) {
            throw new IllegalArgumentException(
                String.format("Illegal method signature, param[%d], String expected,
%s supplied", i, values[i].getClass()));
        }
    }
    ...
```

① method parameters supplied in `Object[]`

② not a specific requirement for this validation — but sanity check we have what is expected

③ names validated must be of type String

12.4. Constraint Validation

Once we have the constraint properly declared and call-correctness validated, the implementation will look similar to most other constraint validations. This method is required to return a true or false.

```
@Override
public boolean isValid(Object[] values, ConstraintValidatorContext context) { ①
    ...
    String name1= (String) values[1];
    String name2= (String) values[2];
    return (StringUtils.isNotBlank(name1) || StringUtils.isNotBlank(name2));
}
```

Chapter 13. Web API Integration

13.1. Vanilla Spring/AOP Validation

From what we have learned in the previous chapters, we know that we should be able to annotate any `@Component` class/interface — including a `@RestController` — and have constraints validated. I am going to refer to this as "Vanilla Spring/AOP Validation" because it is not unique to any component type.

The following snippet shows an example of the Web API `@RestController` that validates parameters according to `Create` and `Default` Groups.

@RestController Validating Constraints using Vanilla AOP Validation

```
@Validated ①
public class ContactsController {
    ...
    @RequestMapping(path=CONTACTS_PATH,
        method= RequestMethod.POST,
        consumes={MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE},
        produces={MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE})
    @Validated(PocValidationGroups.CreatePlusDefault.class) ②
    public ResponseEntity<PersonPocDTO> createPOC(
        @RequestBody
        @Valid ③
        PersonPocDTO personDTO) {
...
```

① triggers validation for component

② configures validator for method constraints

③ identifies constraint for parameter

If we call this with an invalid `personDTO` (relative to the Default or Create groups), we would expect to see validation fail and some sort of error response from the Web API.

13.2. ConstraintViolationException Not Handled

As expected — Spring will validate the constraints and throw a `ConstrainViolationException`. However, Spring Boot — out of the box — does not provide built-in exception advice for `ConstraintViolationException`. That will result in the caller receiving a `500/INTERNAL_SERVER_ERROR` status response with the default error reporting message. It is understandable that it would be the default since constraints can be technically validated and reported from all different levels of our application. The exception could realistically be caused by a real internal server error. However — the reported status does not always have to be generic and misleading.

Default ConstraintViolation API Response

```
> POST http://localhost:64153/api/contacts

{"id":"1","firstName":"Douglass","lastName":"Effertz","dob":[2011,6,14],"contactPoints
":[{"id":null,"name":"Cell","email":"penni.kautzer@hotmail.com","phone":"(876) 285-
7887 x1055","address":{"street":"69166 Angelo Landing","city":"Jaredshire","state":"
IA","zip":"81764-6850"}}]}

< 500 INTERNAL_SERVER_ERROR Internal Server Error ①

{ "url" : "http://localhost:53298/api/contacts",
  "statusCode" : 500,
  "statusName" : "INTERNAL_SERVER_ERROR",
  "message" : "Unexpected Error",
  "description" : "unexpected error executing request:
jakarta.validation.ConstraintViolationException: createPOC.person.id: cannot be
specified for create",
  "timestamp" : "2021-07-01T14:58:48.7772697" }
```

① `INTERNAL_SERVER_ERROR` status is misleading — cause is bad value provided by client

The violation — at least in this case — was a bad value from the client. The `id` property cannot be assigned when attempting to create a contact. Ideally — this would get reported as either a `400/BAD_REQUEST` or `422/UNPROCESSABLE_ENTITY`. Both are 4xx/Client error status and will point to something the client needs to correct.

13.3. ConstraintViolationException Exception Advice

Assuming that the invalid value came from the client, we can map the unhandled `ConstraintViolationException` to a `400/BAD_REQUEST` using (in this case) a global `@RestControllerAdvice`.

The following snippet shows how we can take some of the code we have seen used in the JUnit tests to report validation details — and use that within an `@ExceptionHandler` to extract the details and report as a `400/BAD_REQUEST` to the client.

Mapping ConstraintViolationException to BAD_REQUEST

```
import info.ejava.examples.common.web.BaseExceptionAdvice;
...
@RestControllerAdvice ①
public class ExceptionAdvice extends BaseExceptionAdvice { ②

    @ExceptionHandler(ConstraintViolationException.class)
    public ResponseEntity<MessageDTO> handle(ConstraintViolationException ex) {
        String description = ex.getConstraintViolations().stream()
                .map(v->v.getPropertyPath().toString() + ":" + v.getMessage())
                .collect(Collectors.joining("\n"));
        HttpStatus status = HttpStatus.BAD_REQUEST; ③
```

```
        return buildResponse(status, "Validation Error", description, (Instant)null);
    }
```

① controller advice being applied globally to all controllers in the application context

② extending a class of exception handlers and helper methods

③ hard-wiring the exception to a `400/BAD_REQUEST` status

13.4. ConstraintViolationException Mapping Result

The following snippet shows the Web API response to the client expressed as a `400/BAD_REQUEST`.

ConstraintViolationException Mapped to 400/BAD_REQUEST

```
{ "url" : "http://localhost:53408/api/contacts",
  "statusCode" : 400,
  "statusName" : "BAD_REQUEST",
  "message" : "Validation Error",
  "description" : "createPOC.person.id: cannot be specified for create",
  "timestamp" : "2021-07-01T15:10:59.037162Z" }
```

Converting from a `500/INTERNAL_SERVER_ERROR` to a `400/BAD_REQUEST` is the minimum of what we wanted (at least it is a Client Error status), but we can try to do better. We understood what was requested — but could not process the payload as provided.

13.5. Controller Constraint Validation

To cause the violation to be mapped to a `422/UNPROCESSABLE_ENTITY` to better indicate the problem, we can activate validation within the controller framework itself versus the vanilla Spring/AOP validation.

The following snippet shows an example of the `@RestController` identifying validation and specific validation groups as part of the Web API framework. The `@Validated` annotation is now being used on the Web API parameters.

Activating @RestController Validation of Payload

```
@RequestMapping(path=CONTACTS_PATH,
        method= RequestMethod.POST,
        consumes={MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE},
        produces={MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE})
//@Validated(PocValidationGroups.CreatePlusDefault.class) -- no longer needed ①
public ResponseEntity<PersonPocDTO> createPOC(
        @RequestBody
        //@Valid -- replaced by @Validated ①
        @Validated(PocValidationGroups.CreatePlusDefault.class) ②
        PersonPocDTO personDTO) {
```

① vanilla Spring/AOP validation has been disabled

② Web API-specific parameter validation has been enabled

13.6. MethodArgumentNotValidException

Spring MVC will independently validate the `@RequestBody`, `@RequestParam`, and `@PathVariable` constraints according to internal rules. Spring will throw an `org.springframework.web.bind.MethodArgumentNotValidException` exception when encountering a violation with the request body. That exception is mapped — by default — to return a very terse `400/BAD_REQUEST` response.

The snippet below shows an example response payload for the default `MethodArgumentNotValidException` mapping.

MethodArgumentNotValidException Default Mapping

```
< 400 BAD_REQUEST Bad Request
{"timestamp":"2021-07-01T15:24:44.464+00:00",
 "status":400,
 "error":"Bad Request",
 "message":"",
 "path":"/api/contacts"}
```

By default — we may want to be terse to avoid too much information leakage. However, in this case, let's improve this.

13.7. MethodArgumentNotValidException Custom Mapping

Of course, we can change the behavior if desired using a custom exception handler.

The following snippet shows an example custom exception handler mapping `MethodArgumentNotValidException` to a `422/UNPROCESSABLE_ENTITY`.

MethodArgumentNotValidException Custom Mapping to 422/UNPROCESSABLE_ENTITY

```
@RestControllerAdvice
public class ExceptionAdvice extends BaseExceptionAdvice {
    @ExceptionHandler(ConstraintViolationException.class)
    public ResponseEntity<MessageDTO> handle(ConstraintViolationException ex) { ... }

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<MessageDTO> handle(MethodArgumentNotValidException ex) { ①

        Stream<String> fieldMsgs = ex.getFieldErrors().stream() ②
            .map(e -> e.getObjectName()+"."+e.getField()+": "+e.getDefaultMessage());
        Stream<String> globalMsgs = ex.getGlobalErrors().stream() ③
            .map(e -> e.getObjectName() +": "+ e.getDefaultMessage());
```

```
        String description = Stream.concat(fieldMsgs, globalMsgs)
            .collect(Collectors.joining("\n"));
        return buildResponse(HttpStatus.UNPROCESSABLE_ENTITY, "Validation Error",
            description, (Instant)null);
    }
```

① Spring MVC throws `MethodArgumentNotValidException` for `@RequestBody` violations

② reports fields of objects in error

③ reports overall objects (e.g., cross-parameter violations) in error

13.7.1. MethodArgumentNotValidException Custom Mapping Response

This results in the client receiving an HTTP status indicating the request was understood, but the payload provided was invalid. The description is as terse or verbose as we want it to be.

MethodArgumentNotValidException Custom Mapping Response

```
{ "url" : "http://localhost:53818/api/contacts",
  "statusCode" : 422,
  "statusName" : "UNPROCESSABLE_ENTITY",
  "message" : "Validation Error",
  "description" : "personPocDTO.id: cannot be specified for create",
  "timestamp" : "2021-07-01T15:38:48.045038Z" }
```

Can Also Supply Client Value if Permitted

The exception handler has access to the invalid value if security policy allows information like that to be in the response. Note that error messages tend to be placed into logs and logs can end up getting handled at a generic level. For example, you would not want an invalid partial but mostly correct SSN to be part of an error log.

13.8. @PathVariable Validation

Note that the Web API maps the `@RequestBody` constraint violations independently of the other parameter types.

The following snippet shows an example of validation constraints applied to `@PathVariable`. These are physically in the URI.

@PathVariable Validation

```
@RequestMapping(path= CONTACT_PATH,
        method=RequestMethod.GET,
        produces={MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE})
public ResponseEntity<PersonPocDTO> getPOC(
        @PathVariable(name="id")
        @Pattern(regexp = "[0-9]+", message = "must be a number") ①
```

```
                String id) {
```

① validation here is thru vanilla Spring/AOP validation

13.9. @PathVariable Validation Result

The Web API (using vanilla Spring/AOP validation here) throws a `ConstraintViolationException` for `@PathVariable` and `@RequestParam` properties. We can leverage the custom exception handler we already have in place to do a decent job reporting status.

The following snippet shows an example response that is being mapped to a `400/BAD_REQUEST` using our custom exception handler for `ConstraintViolationException`. `400/BAD_REQUEST` seems appropriate because the `id` path parameter is invalid garbage (`1⋯34`) in this case.

@PathVariable Validation Violation Response

```
> HTTP GET http://localhost:53918/api/contacts/1...34, headers={masked}
< BAD_REQUEST/400
{ "url" : "http://localhost:53918/api/contacts/1...34",
  "statusCode" : 400,
  "statusName" : "BAD_REQUEST",
  "message" : "Validation Error",
  "description" : "getPOC.id: must be a number",
  "timestamp" : "2021-07-01T15:51:34.724036Z" }
```

Remember — if we did not have that custom exception handler in place for `ConstraintViolationException`, the HTTP status would have been a `500/INTERNAL_SERVER_ERROR`.

Unmapped ConstraintViolationException for @PathVariable Violation

```
< 500 INTERNAL_SERVER_ERROR Internal Server Error
{"timestamp":"2021-07-01T19:21:31.345+00:00","status":500,"error":"Internal Server
Error","message":"","path":"/api/contacts/1...34"}
```

13.10. @RequestParam Validation

`@RequestParam` validation follows the same pattern as `@PathVariable` and gets reported using a `ConstraintViolationException`.

@RequestParam Validation

```
@RequestMapping(path= EXAMPLE_CONTACTS_PATH,
        method=RequestMethod.POST,
        consumes={MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE},
        produces={MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE})
public ResponseEntity<PersonsPageDTO> findPocsByExample(
        @RequestParam(value = "pageNumber", defaultValue = "0", required = false)
        @PositiveOrZero
```

```
        Integer pageNumber,

        @RequestParam(value = "pageSize", required = false)
        @Positive
        Integer pageSize,

        @RequestParam(value = "sort", required = false) String sortString,

        @RequestBody PersonPocDTO probe) {
```

13.11. @RequestParam Validation Violation Response

The following snippet shows an example response for an invalid set of query parameters.

@RequestParam Validation Violation Response

```
> POST http://localhost:53996/api/contacts/example?pageNumber=-1&pageSize=0
{ ... }
> BAD_REQUEST/400
{ "url" : "http://localhost:53996/api/contacts/example?pageNumber=-1&pageSize=0",① ②
  "statusCode" : 400,
  "statusName" : "BAD_REQUEST",
  "message" : "Validation Error",
  "description" : "findPocsByExample.pageNumber: must be greater than or equal to 0
\nfindPocsByExample.pageSize: must be greater than 0",
  "timestamp" : "2021-07-01T15:55:44.089734Z" }
```

① pageNumber has an invalid negative value

② pageSize has an invalid non-positive value

13.12. Non-Client Errors

One thing you may notice with the previous examples is that every constraint violation was blamed on the client — whether it was bad server code calling internally or not.

As an example, let's have the API require that `value` be non-negative. A successful validation of that constraint will result in a service method call.

Web API Requires Client Supply Non-Negative @RequestParam

```
@RequestMapping(path = POSITIVE_OR_ZERO_PATH,
method=RequestMethod.GET,
    produces = {MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE})
public ResponseEntity<?> positive(
        @PositiveOrZero ①
        @RequestParam(name = "value") int value) {
    PersonPocDTO resultDTO = contactsService.positiveOrZero(value); ②
```

① @RequestParam validated

② `value` from valid request passed to service method

13.13. Service Method Error

The following snippet shows that the service call makes an obvious error by passing the `value` to an internal component requiring the value to not be positive.

Downstream Component Makes Error

```
public class PocServiceImpl implements PocService {
    ...
    public PersonPocDTO positiveOrZero(int value) {
        //obviously an error!!
        internalComponent.negativeOrZero(value);
        ...
```

The internal component leverages the Bean Validation by placing a `@NegativeOrZero` constraint on the `value`. This is obviously going to fail when the value is ever non-zero.

Internal Component Declares Violated Constraint

```
@Component
@Validated
public class InternalComponent {
    public void negativeOrZero(@NegativeOrZero int value) {
```

13.14. Violation Incorrectly Reported as Client Error

The snippet below shows an example response of the internal error. It is being blamed on the client — when it was actually an internal server error.

Internal Server Error Incorrectly Reported as Client Error

```
> GET http://localhost:54298/api/contacts/positiveOrZero?value=1

< 400 BAD_REQUEST Bad Request
{ "url":"http://localhost:54298/api/contacts/positiveOrZero?value=1",
  "statusCode":400,
  "statusName":"BAD_REQUEST",
  "message":"Validation Error",
  "description":"negativeOrZero.value: must be less than or equal to 0",
  "timestamp":"2021-07-01T16:23:27.666154Z"}
```

13.15. Checking Violation Source

One thing we can do to determine the proper HTTP response status — is to inspect the source

information of the violation.

The following snippet shows an example of inspecting whether the violation was reported by a class annotated with `@RestController`. If from the API, then report the `400/BAD_REQUEST` as usual. If not, report it as a `500/INTERNAL_SERVER_ERROR`. If you remember — that was the original default behavior.

Internal Error Detected through Source Information

```
@ExceptionHandler(ConstraintViolationException.class)
public ResponseEntity<MessageDTO> handle(ConstraintViolationException ex) {
    String description = ...

    boolean isFromAPI = ex.getConstraintViolations().stream() ①
            .map(v -> v.getRootBean().getClass().getAnnotation(RestController.class))
            .filter(a->a!=null)
            .findFirst()
            .isPresent();

    HttpStatus status = isFromAPI ?
            HttpStatus.BAD_REQUEST : HttpStatus.INTERNAL_SERVER_ERROR;
    return buildResponse(status, "Validation Error", description, (Instant)null);
}
```

① `isFromAPI` set to true if any of the violations came from a component annotated with `@RestController`

13.16. Internal Server Error Correctly Reported

The following snippet shows the response to the client when our exception handler detects that is handling at least one violation generated from a class annotated with `@RestController`.

Internal Server Error Correctly Reported

```
{ "url" : "http://localhost:54434/api/contacts/positiveOrZero?value=1",
  "statusCode" : 500,
  "statusName" : "INTERNAL_SERVER_ERROR",
  "message" : "Validation Error",
  "description" : "negativeOrZero.value: must be less than or equal to 0",
  "timestamp" : "2021-07-01T16:45:50.235724Z" }
```

Any Source of Constraint Violation May be used to Impact Behavior

There is no magic to using the `@RestController` annotation as a trigger for certain behavior. Annotations are used all the time to denote classes of a certain pattern. One could create a custom annotation that explicitly indicates what we are looking to identify.

Limit Externalizing Internal Implementation Details

> The example provided internal server error details that could be of no use to the client and could be exposing internal implementation details. Strongly consider placing the details in a server-side log and state the basics to the client.

13.17. Service-detected Client Errors

Assuming we do a thorough job validating all client inputs at the `@RestController` level, we might be done. However, what about the case where the client validation is pushed down to the `@Service` components. We would have to adjust our violation source inspection.

The following snippet shows an example of a service validating client requests using the same constraints as before — except this is in a lower-level component.

Service Validating Client Request

```
public interface PocService {
    @NotNull
    @Validated(PocValidationGroups.CreatePlusDefault.class)
    PersonPocDTO createPOC(
        @NotNull
        @Valid PersonPocDTO personDTO);
```

Without any changes, we get violations reported as `400/BAD_REQUEST` status — which, as I stated in the beginning, was "OK".

Service Violation Reported as 400/BAD_REQUEST

```
< 400 BAD_REQUEST Bad Request
{ "url" : "http://localhost:55168/api/contacts",
  "statusCode" : 400,
  "statusName" : "BAD_REQUEST",
  "message" : "Validation Error",
  "description" : "createPOC.person.id: cannot be specified for create",
  "timestamp" : "2021-07-01T17:40:12.221497Z" }
```

I won't try to improve the HTTP status using source annotations on the validating class. I have already shown how to do that. Let's try another technique.

13.18. Payload

One other option we have to is leverage the payload metadata in each annotation. `Payload` classes are interfaces extending `jakarta.validation.Payload` that identify certain characteristics of the constraint.

Annotations Include Payload Metadata

```
public @interface Xxx {
    String message() default "...";
```

```
    Class<?>[] groups() default { };
    Class<? extends Payload>[] payload() default { }; ①
```

① Annotations can carry extra metadata in the payload property

The snippet below shows an example of a Payload subtype that expresses the violation should be reported as a `500/INTERNAL_SERVICE_ERROR`.

Example HttpStatus Payload

```
public interface InternalError extends Payload {}
```

This payload information can be placed in constraints that are known to be validated by internal components.

Internal Component with Payload

```
@Component
@Validated
public class InternalComponent {
    public void negativeOrZero(@NegativeOrZero(payload = InternalError.class) int
value) {
```

13.19. Exception Handler Checking Payloads

The snippet below shows our generic, global advice factoring in whether the violation came from an annotation with an `InternalError` in the payload.

Exception Handler Checking Payloads

```
@ExceptionHandler(ConstraintViolationException.class)
public ResponseEntity<MessageDTO> handle(ConstraintViolationException ex) {
    String description = ...;
    boolean isFromAPI = ...;

    boolean isInternalError = isFromAPI ? false : ①
        ex.getConstraintViolations().stream()
            .map(v -> v.getConstraintDescriptor().getPayload())
            .filter(p-> p.contains(InternalError.class))
            .findFirst()
            .isPresent();

    HttpStatus status = isFromAPI || !isInternalError ?
            HttpStatus.BAD_REQUEST : HttpStatus.INTERNAL_SERVER_ERROR;

    return buildResponse(status, "Validation Error", description, (Instant)null);
}
```

① `isInternalError` set to true if any violations contain the `InternalError` payload

13.20. Internal Violation Exception Handler Results

The following snippet shows an example of a constraint violation where none of the violations where assigned a payload with `InternalError`. The status is returned as `400/BAD_REQUEST`.

Violation From Annotation without Payload

```
> POST http://localhost:55288/api/contacts
< 400/BAD_REQUEST
  "url" : "http://localhost:55288/api/contacts",
  "statusCode" : 400,
  "statusName" : "BAD_REQUEST",
  "message" : "Validation Error",
  "description" : "createPOC.person.id: cannot be specified for create",
  "timestamp" : "2021-07-01T17:56:23.080884Z"
}
```

The following snippet shows an example of a constraint violation where at least one of the violations were assigned a payload with `InternalError`. The client may not be able to make heads-or-tails out of the error message, but at least they would know it is something on the server-side to be corrected.

Violation from Annotation with InternalError Payload

```
> GET http://localhost:57547/api/contacts/positiveOrZero?value=1
< INTERNAL_SERVER_ERROR/500
{ "url" : "http://localhost:57547/api/contacts/positiveOrZero?value=1",
  "statusCode" : 500,
  "statusName" : "INTERNAL_SERVER_ERROR",
  "message" : "Validation Error",
  "description" : "negativeOrZero.value: must be less than or equal to 0",
  "timestamp" : "2021-07-01T20:25:05.188126Z" }
```

Chapter 14. JPA Integration

Bean Validation is integrated into the JPA standard. This can be used to validate entities mostly when created, updated, or deleted. Although not part of the standard, it is also used by some providers to customize generated database schema with additional RDBMS constraints (e.g., `@NotNull`, `@Size`). By default, the JPA provider will implement validation of the `Default` group for all `@Entity` classes during creation or update.

The following is a list of JPA properties that can be used to impact the behavior. They all need to be prefixed with `spring.jpa.properties.` when using Spring Boot properties to set the value.

Table 3. JPA Validation Configuration Properties

`jakarta.persistence.validation.mode`	ability to control validation at a high level	• **auto** - implement validation if provider available **(default)** • **callback** - validation is required and will fail if the provider is missing • **none** - disable validation entirely within JPA
`jakarta.persistence.validation.group.pre-persist`	identify groups(s) validated before inserting new row	• jakarta.validation.groups.Default.class **(default)**
`jakarta.persistence.validation.group.pre-update`	identify group(s) validated before updating existing row	• jakarta.validation.groups.Default.class **(default)**
`jakarta.persistence.validation.group.pre-remove`	identify group(s) validated before removing existing row	• (none) **(default)**

Chapter 15. Mongo Integration

A basic Bean Validation implementation is integrated into Spring Data Mongo. It leverages event-specific callbacks from `AbstractMongoEventListener`, which is integrated into the Spring `ApplicationListener` framework.

There are no configuration settings, and after you see the details — you will quickly realize that they mean to handle the most common case (validate the `Default` group on `save()`) and for us to implement the corner-cases.

The following snippet shows an example of activating the default MongoRepository validation.

Basic MongoRepository Validation Configuration

```
import
org.springframework.data.mongodb.core.mapping.event.ValidatingMongoEventListener;
...
@Configuration
public class MyMongoConfiguration {
    @Bean
    public ValidatingMongoEventListener mongoValidator(Validator validator) {
        return new ValidatingMongoEventListener(validator);
    }
}
```

15.1. Validating Saves

To demonstrate validation within the data tier, let's assume that our document class has a constraint for the `dob` to be supplied.

Example Mongo Document Class with Constraint

```
@Document(collection = "pocs")
public class PersonPOC {
    ...
    @NotNull
    private LocalDate dob;
    ...
}
```

When we attempt to save a PersonPOC in the repository without a `dob`, the following example shows that the Java source object is validated, a violation is detected, and a `ConstraintViolationException` is thrown.

Example Validation on save()

```
//given
PersonPOC noDobPOC = mapper.map(pocDTOFactory.make().withDob(null));
```

```
//when
assertThatThrownBy(() -> contactsRepository.save(noDobPOC))
        .isInstanceOf(ConstraintViolationException.class)
        .hasMessageContaining("dob: must not be null");
```

There is nothing more to it than that until we look into the implementation of `ValidatingMongoEventListener`.

15.2. ValidatingMongoEventListener

`ValidatingMongoEventListener` extends `AbstractMongoEventListener`, has a `Validator` from injection, and overrides a single event callback called `onBeforeSave()`.

ValidatingMongoEventListener

```
package org.springframework.data.mongodb.core.mapping.event;
...
public class ValidatingMongoEventListener extends AbstractMongoEventListener<Object> {
    ...
    private final Validator validator;
    @Override
    public void onBeforeSave(BeforeSaveEvent<Object> event) {
        ...
    }
}
```

It takes little imagination to guess how the rest of this works. I have removed the debug code from the method and provided the remaining details here.

onBeforeSave Details

```
@Override
public void onBeforeSave(BeforeSaveEvent<Object> event) {
    Set violations = validator.validate(event.getSource());

    if (!violations.isEmpty()) {
        throw new ConstraintViolationException(violations);
    }
}
```

The `onBeforeSaveEvent` is called after the source Java object has been converted to a form that is ready for storage.

15.3. Other AbstractMongoEventListener Events

There are many reasons — beyond validation, (e.g., sub-document ID generation) — we can take advantage of the `AbstractMongoEventListener` callbacks, so it will be good to provide an overview of them now.

- There are three core events: Save, Load, and Delete
- Several **possible** stages to each core event
  - before action performed (e.g., delete)
    - and before converting between a Java object and `Document` (e.g., save and load)
    - and after converting between a Java object and `Document` (e.g., save and load)
  - after action is complete (e.g., save)

The following table lists the specific events.

Table 4. MongoMappingEvents

onApplicationEvent(MongoMappingEvent<?> event)	general purpose event handler
onBeforeConvert(BeforeConvertEvent<E> event)	callback before a Java object converted to `Document`
onBeforeSave(BeforeSaveEvent<E> event)	callback after Java object converted to `Document` and before saved to DB
onAfterSave(AfterSaveEvent<E> event)	callback after `Document` saved
onAfterLoad(AfterLoadEvent<E> event)	callback after `Document` loaded from DB and before converted to a Java object
onAfterConvert(AfterConvertEvent<E> event)	callback after `Document` converted to Java object
onBeforeDelete(BeforeDeleteEvent<E> event)	callback before document deleted from DB
onAfterDelete(AfterDeleteEvent<E> event)	callback after document deleted from DB

15.4. MongoMappingEvent

The `MongoMappingEvent` itself has three main items.

- Collection Name — name of the target collection
- Source — the source or target Java object
- `Document` — the source or target `bson` data type stored to the database

Our validation would always be against the `source`, so we just need a callback that provides us with a read-only value to validate.

Chapter 16. Patterns / Anti-Patterns

Every piece of software has an interface with some sort of pre-conditions and post-conditions that have some sort of formal or informal constraints. Constraint validation — whether using custom code or Bean Validation framework — is a decision to be made when forming the layers of a software architecture. The following patterns and anti-patterns list a few concerns to address. The original outline and content provided below is based on Tom Hombergs' Bean Validation Anti-Patterns article.

16.1. Data Tier Validation

The Data tier has long been the keeper of data constraints — especially with RDBMS schema.

- Should the constraint validations discussed be implemented at that tier?
- Can validation wait all the way to the point where it is being stored?
- Should service and other higher levels of code be working with data that has not been validated?

16.1.1. Data Tier Validation Safety Checks

Hombergs' suggestion was to use the data tier validation as a safety check, but not the only layer. [1]

That, of course, makes a lot of sense since the data tier may not need to know what a valid e-mail looks like or (to go a bit further) what type of e-mail addresses we accept? However, the data tier will want to sanity check that required fields exist and may want to go as far as validating format if query implementations require the data to be in a specific form.

16.2. Use case-specific Validation

Re-use is commonly a goal in software development. However, as we saw with validation groups — some data types have use case-specific constraints.

The simple example is when `id` could not be provided during a create but was legal in all other situations.

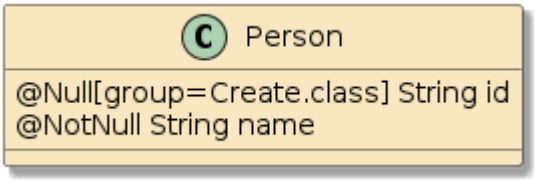


Figure 3. Re-usable Data Class with Use case-Specific Semantics

As more use case-specific constraints pile up on re-usable classes, they can get very cluttered and present a violation of single purpose Single-responsibility principle.

16.2.1. Separate Syntactic from Semantic Validation

Hombergs proposes we

- use Bean Validation for syntactical validation for re-usable data classes
- implement query methods in the data classes for semantic state and perform checks against that specific state within the use case-specific code. [1]

One way of implementing use case-specific query methods and having them leverage Bean Validation constraints and a re-used data type would be to create use case-specific decorators or wrappers. Lombok's experimental `@Delegate` code generation may be of assistance here.

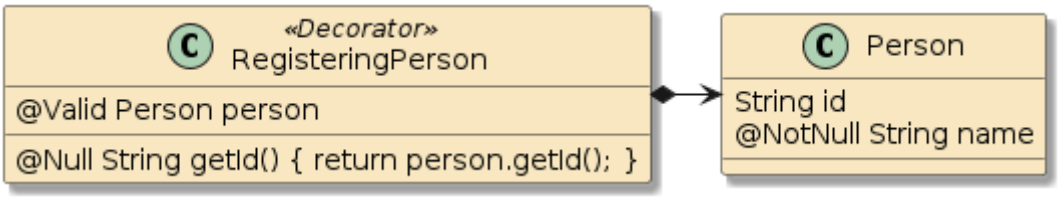


Figure 4. Use case-Specific Data Wrapper

16.3. Anti: Validation Everywhere

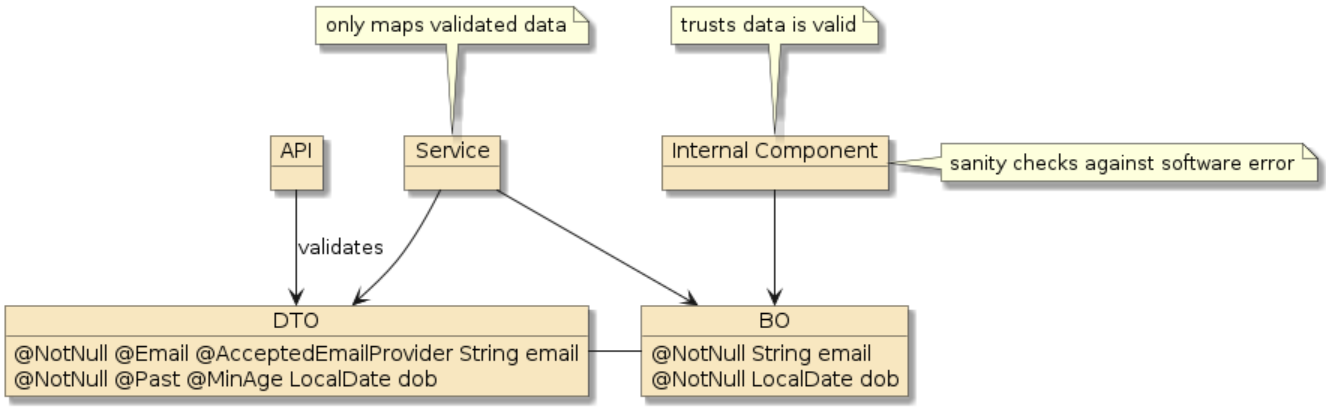
It is likely for us to want to validate at the client interface (Web API) since these are very external inputs. It is also likely for us to want to validate at the service level because our service could be injected into multiple client interfaces. It is then likely that internal components see how easy it is to add validation triggers and add to the mix. At the end of the line — the persistence layer adds a final check.

In some cases, we can get the same information validated several times. We have already shown in the Bean Validation details earlier in this topic — the challenge can be to determine what is a client versus internal issue when a violation occurs.

16.3.1. Establish Validation Architecture

Hombergs recommends having a clear validation strategy versus ad-hoc everywhere [1]

I agree with that strategy and like to have a clear dividing line of "once it reaches this point — data its valid". This is where I like to establish service entry points (validated) and internal components (sanity checked). Entry points check everything about the data. Internal components trust that the data given is valid and only need to verify if a programming error produced a null or some other common illegal value.



I also believe separating data types into external ("DTOs") and internal ("BOs") helps thin down the concerns. DTO classes would commonly be thorough and allow clients to know exactly what constraints exist. BO classes — used by the business and persistence logic only accept valid DTOs and should be done with detailed validation by the time they are mapped to BO classes.

16.3.2. Separating Persistence Concerns/Constraints

Hombergs went on to discuss a third tier of data types — persistence tier data types — separate from BOs as a way of separating persistence concerns away from BO data types. [2] This is part of implementing a Hexagonal Software Architecture where the core application has no dependency on any implementation details of the other tiers. This is more of a plain software architecture topic than specific to validation — but it does highlight how there can be different contexts for the same conceptual type of data processed.

[1] *"Bean Validation Anti-Patterns"*, Tom Hombergs

[2] *"Get Your Hands Dirty on Clean Architecture"*, Tom Hombergs, 2019

Chapter 17. Summary

In this module, we learned:

- to add Bean Validation dependencies to the project
- to add declarative pre-conditions and post-conditions to components using the Bean Validation API
- to define declarative validation constraints
- to configure a `ValidatorFactory` and obtain a `Validator`
- to programmatically validate an object
- to programmatically validate parameters to and response from a method call
- to enable Spring/AOP validation for components
- to implement custom validation constraints
- to implement a cross-parameter validation constraint
- to configure Web API constraint violation responses
- to configure Web API parameter validation
- to configure JPA validation
- to configure Spring Data Mongo Validation
- to identify some patterns/anti-patterns for using validation