International Journal of Computer Trends and Technology
ISSN: 2231-2803 / https://doi.org/10.14445/2231280453/IJCTT-V67I3P121
Volume 67 Issue 3,109-112, March 2019

Original Article

Transform Dynamic Validation Metadata to Runtime Java Bean Validation Constraint Annotations

Vijay Kumar Pandey

Director of Technology Solutions, Intueor Consulting, In. Irvine, CA (United States of America)

Abstract *- The paper is intended to provide an in-depth understanding of how to transform dynamic validation metadata to a runtime java class(es) with relevant fields annotated with proper bean validation annotations, which are dynamically generated based on the validation metadata. This dynamic class generation is performed through an open-source library, Byte Buddy (which depends on ASM - ASM is an all-purpose Java bytecode manipulation and analysis framework). This understanding will help developers build dynamic validation features in their system while validating the properties of a dynamic screen built with the help of dynamic properties metadata. The intended audience for this article includes application architects, software designers, and software programmers who participate in the design, architecting, and development of robust and complex enterprise-wide applications based on Java and Jakarta EE (or spring) and want to utilize the Java Bean Validation (1.1 or 2) in dynamic scenarios.*

Keywords *- Java Bean Validation (BV), Byte Buddy, ASM, Jakarta EE.*

I. INTRODUCTION

Bean Validation Constraints are defined by combining a constraint annotation and a list of constraint validation implementations. The constraint annotation is applied to types, fields, methods, constructors, parameters, or other constraint annotations in the case of composition. A constraint on a JavaBean is expressed through one or more annotations. An annotation is considered a constraint definition if its retention policy contains *RUNTIME* and if the annotation itself is annotated with *javax. Validation.Constraint*. Every constraint annotation must define a *message* element of type *String*.

II. CONSTRAINT DEFINED PROPERTIES

A constraint definition may have attributes specified when the constraint is applied to a JavaBean. The properties are mapped as annotation elements. The annotation element names *message*, *groups*, *validation applies to*, and *payload* are considered reserved names; annotation elements starting with valid are not allowed; a constraint may use any other element name for its attributes. The constraint annotation definitions may define additional elements to parameterize the constraint. For example, a constraint that validates the length of a string can use an annotation element named *length* to specify the maximum length when the constraint is declared.

A. Constraint Validation

A constraint validation implementation performs the validation of a given constraint annotation for a given type. The implementation classes are specified by the *validated* element of the *@Constraint* annotation that decorates the constraint definition. The constraint validation implementation implements the *ConstraintValidator* interface.

B. Constraint Configuration

Bean Validation provides two ways to manage constraints metadata. The first is through the annotations, as already discussed, and the other is through *XML* constraint mapping configuration that can override the annotation metadata for certain classes. By default, all constraint declarations expressed via annotation are ignored for classes described in XML. Bean Validation can force to consider both annotations and XML constraint declarations by using *ignore-annotations="false"* on the bean.

Some of the important Bean Validation API classes are:

- *javax.validator.ValidatorFactory* – Factory returning initialized *Validator* instances.
- *javax.validator.Validator* – Validates bean instances.

C. Dynamic Runtime Class

Based on the validation metadata, Byte Buddy will be used to create run time classes mainly at the application start-up, which will have constraint annotations either on the field or property. Then these classes will be used to validate the value through the *Validator* interface dynamically. *Validator* provides the following method to validate a value.

```
<T>
Set<ConstraintViolation<T>>validateValue(Class<T>beanType,
String propertyName, Object value, Class<?>... groups)
```

- *First argument "*Class<T>*beanType"* – This is the run time class created during application start-up through *Byte Buddy* and has the relevant property name, which would have the constraint annotations to be validated.
- *The second argument, "String property Name,"* – Is the name of the property that will be validated and must be present in the class defined above by the first argument.
- *Third argument "Object value"* – This is the value that will be validated against the constraints defined on the property.
- *Fourth argument "Class*<?>… groups" – Constraints can belong to one or more groups or contexts. Applying a subset of the constraints for a given use case is useful. By default, the *Default* group is used.

D. Dynamic Validation Metadata

Dynamic Validation Metadata can be present in a database or another store, fetched during application start-up to generate run time Bean Validation classes. To showcase the behavior, two different standard constraints are chosen: *javax.validation.constraints.NotNull* and *javax.validation.constraints.Max*.

***First Property** ValidationMetaData*

Property Name:property1 (could be any valid java property name)
Property Type: java.lang.String(can be any type as per the requirement)
Constraint Name: javax.validation.constraints.NotNull
Message: I will use the default message
Payload: default
Groups: default

***Second Property** ValidationMetaData*

Property Name: property2 (could be any valid java property name)
Property Type: java.lang.Long(has to be one of the types supported by this *Max* Constraint)
Constraint Name: javax.validation.constraints.Max
Message: I will use the default message
Payload: default
Groups: default
Value: 100 (any max value for this constraint)

In the above metadata, two properties validation metadata has been showcased. These properties could have been used on a dynamic screen in the real-world application, which needs to be validated against Bean Validation. For these scenarios, the default capability of defining constraint annotations on static java classes or in the xml will not suffice. This is where the feature of defining constraints during runtime class generation will be of real value for these dynamic scenarios.

E. Annotate Runtime Classes

To easily identify (dynamically)generated bean validation runtime classes, they can be annotated(not a mandatory step) with a marker annotation (no attributes). Let this annotation be *"com.vp.bv.BVRuntimeAnnt"*. This annotation class can be created manually.

```
package com.vp.bv;

import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

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

@Target({ TYPE })
@Retention(RUNTIME)
@Documented
public @interface BVRuntimeModelType {
}
```

F. Byte Buddy

Byte Buddy is a code generation and manipulation library for creating and modifying Java classes during the runtime of a Java application and without the help of a compiler. Byte Buddy is used to generating these run time classes.

III.BYTE BUDDY RUNTIME CLASS GENERATION

A. Annotation Description

The first step of generating the classes is to create the *AnnotationDescription*. As discussed earlier, this will be used to annotate the actual runtime class.

```
AnnotationDescriptionannDesc =
AnnotationDescription.Builder.ofType(BVRuntimeModelType.cla
ss).build();
```

B. DynamicTypeBuilder

The main entry point for creating the dynamic runtime classes through Byte Buddy is the *ByteBuddy* and *DynamicType.Builder* class. The below code is a snapshot of creating an empty runtime class of a given name and annotated with a given type.

```
Builder<?>builder = new
ByteBuddy().subclass(Object.class).name(fullyQualifiedClassNam
e).annotateType(annDesc)

*fullyQualifiedClassName – The full name of the dynamic class to
build. For this case, the name will be "com.vp.bv.DynaBv1"
* ann desk – It's the AnnotationDescription that was created earlier.
```

C. Dynamic Class Field Creation

After creating the empty dynamic class in the above step, the below code will add fields based on the validation metadata details provided earlier.

FieldDefinition.Optional.Valuable<?>fieldDefBuilder = builder.**defineField**(fieldName,fieldType, Visibility.PRIVATE);

*fieldName – This will come from the validation metadata such as "property1" and "property2."
* fieldType – This will come from the validation metadata such as "String" and "Long."

D. Constraint Annotation Creation

After creating the fields in the dynamic class, create actual constraint annotations, which will then be annotated on the fields.

Class<? extends Annotation>annotationType = (Class<? extends Annotation>) Class.forName(constraintClassName, true,classLoader);

*constraintClassName – This will come from the validation metadata such as "javax.validation.constraints.NotNull" and "javax.validation.constraints.Max"

Use the above type to create the Byte Buddy annotation builder AnnotationDescription.BuilderanntBuilder = AnnotationDescription.Builder.**ofType**(annotationType);

E. Constraint Annotation Property Creation

After creating the empty constraint annotation builder, the required annotation property will be set with the values from the validation metadata. The code below shows the mechanism to set any reserved property names of the constraint annotation or any custom property from the validation metadata.

anntBuilder.**define**("message", "{dynamic_message}");

*{dynamic_message} – If there was some dynamic message associated with the constraint annotation, it could be set as above

anntBuilder.**define**("value", 100);

*value – The above "value" property needs to be set for the constraint annotation "javax.validation.constraints.Max", the actual value (100) will come from the validation metadata.

F. Annotate Field

Once the *AnnoationDescription.The builder* has been created, and it is necessary to generate the actual ByteBuddy*AnnotationDescription* before these constraint annotations can be annotated on the fields created earlier.

AnnotationDescriptionanntDesc= anntBuilder.**build**();

*If a field has multiple annotations, each will be built and added to a List, which can then be used to annotate the field.

Annotate Field:

builder = fieldDefBuilder.**annotateField**(anntDescList.toArray(new AnnotationDescription[0]))

*fieldDefBuilder – The field builder that was created in Step C

G. Runtime Class Generation

Once the fields have been created and annotated with the constraint annotations, the dynamic builder is ready to create the run time class.

Step 1: Unloaded<?>unloadedType = builder.**make**();

Step 2: Class<?>dynamicType = unloadedType.**load**(classLoader, ClassLoadingStrategy.Default.WRAPPER).getLoaded();

*dynamic type – This is the actual java class loaded in the Java Virtual Machine (JVM). This class is available with the JVM to be used further for Bean Validation from this point onwards.

Note: In a real-world application, if, let's say, there are 1000 dynamic property/fields with bean validation metadata, it might be better to divide the number of fields in each class, maybe around 25, and that will lead to 40 classes, each having 25 fields annotated with bean validation constraints. These loaded classes can be stored in some map in the servlet context or other application scoped bean for easier access during bean validation.

H. Runtime Class Inspection & Debug

During development or even during production, the development team may need to debug any issues in the generated runtime class so that the class created can be written to a file system. This could then be decompiled to understand the actual source code better.

unloadedType.**saveIn**(baseDirectoryPath);

*base directory path – The base path where actual class files will be written could be decompiled with tools like JD Decompiler to have a peek of the source code.

Below source code (class DynaBv1)as seen through decompiler for all the above steps**:**

```
package com.vp.bv;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Max;

@BVRuntimeModelType
public class DynaBv1 {

@NotNull(message="{javax.validation.constraints.NotNull.message}", payload={}, groups={})
    private String property1;


@Max(message="{javax.validation.constraints.Max.message}", value = 100, payload={}, groups={})
    private Long property2;
}
```

IV. BEAN VALIDATION – DYNAMIC RUNTIME CLASSES

Dynamic runtime generated classes can be saved in some context to be easily available wherever bean validation for such dynamic scenarios is needed.

```
List<Class<?>>dynaRuntimeClassList = //from Step III above – //generated list of runtime classes

Map<String, String>dynaBVFieldClassMap = new HashMap<>();
    for (Class<?>clazz :dynaRuntimeClassList) {
            Field[] fields = clazz.getDeclaredFields();
            for (Field field : fields) {
            dynaBVFieldClassMap.put(field.getName(), clazz.getName());
            }
}
```

```
*Store the dynaBVFieldClassMap, maybe in ServletContext (if web application), or some application scoped CDI bean or some singleton for easy reference.
```

A. Bean Validation Invocation

Once the dynamic runtime classes are generated and saved in some context for easy access, the following code sample shows how to validate the property values. The values will be validated against the constraint annotation annotated on the field in the dynamic classes. If the constraint fails, it will return a set of unique constraint violations.

```
javax.validation.Validator validator = //find the validator;
Class<?>[] valGroupsArray = //find the validator groups;

//get the dynamic runtime class for the property which is validated
Class<?>dynaBvClass =
dynaBVFieldClassMap.get(propertyName);

//invoke the actual validation returning any violations
Set<ConstraintViolation<?>>violationsActual =
validator.validateValue(dynaBvClass,
propertyName,propertyValue, valGroupsArray);
```

V. CONCLUSION

This paper presents a unique approach to performing Bean Validation for dynamic properties, whose validation metadata is stored in some form of storage such as database, XML, and others. One of the most common uses will be for the enterprise applications having dynamic screens built during runtime and have no way to use the standard Bean Validation for their dynamic properties. This paper presents the inner workings of generating the dynamic runtime classes through Byte Buddy and using those classes to perform the actual validation for the dynamic properties. Bean Validation 1.1 is part of JEE 7, and Bean Validation 2.0 is part of JEE 8.

REFERENCES

[1] Bean Validation 1.1 JSR 349 - https://beanvalidation.org/1.1/
[2] Bean Validation 2.0 JSR 380- https://beanvalidation.org/2.0/
[3] Byte Buddy - https://bytebuddy.net
[4] ASM - https://asm.ow2.io/
[5] Jakarta Enterprise Edition (JEE) - https://jakarta.ee/
[6] JEE 7 Specification - https://jcp.org/en/jsr/detail?id=342
[7] JEE 8 Specification - https://jcp.org/en/jsr/detail?id=366