1. Specification “Bean Validation in JAXB”

1.1. Brief Description

This specification defines Bean Validation in JAXB facility. Such a facility adds support for Bean Validation to the marshalling framework, schema compiler and schema generator. JavaBeans can be enhanced with validation metadata called BV constraints, which imposes rules upon the object instances of the Bean. The object instances will be automatically validated during marshalling and unmarshalling processes. Enhancements to schema compiler and schema generator will handle transformation and/or preservation of validation metadata throughout JavaBeans lifecycle.

1.2. Target

This facility is intended to become a standalone specification, untied to Java™ SE or Java™ EE platform. Its target platform is Java™ SE version 1.7. The facility is an enhancement to object-xml binding framework JAXB version 2 (part of Java SE platform since version 1.6). The goal is to provide support for automatical validation of JavaBeans using Bean Validation. Both of the currently available versions of Bean Validation, 1.0 (part of Java EE 6 platform) and 1.1 (part of Java EE 7 platform) will be fully supported.

1.3. What needs are going to be addressed?

The proposed specification will vastly simplify the creation and maintenance of XML-enabled Java programs with automatic validation of object representations of the marshalled and unmarshalled XML documents. It presents a standard for translation of XML Restrictions and Facets to Bean Validation constraints and vice-versa. It will enable users to focus on business logic and will automatically convert and preserve validation metadata between XML and Java worlds and thus another milestone in the mission of resolving Object-to-XML impedance mismatch will be reached.

1.4. Why isn't this need met by existing specifications?

The only existing solution for integrating JAXB and Bean Validation are a few XJC plugins that only allow for generation of annotations on schema-derived compiled classes. While it is possible to write XML-enabled programs with manually invoked Bean Validation support, it is not possible to write programs that would invoke Bean Validation automatically during marshalling and unmarshalling events. Also, all validation metadata is lost after schema generation process.

1.5. Detailed description

The proposed specification describes enhancements to three JAXB components: the marshalling framework, schema compiler and schema generation. It also illustrates facility-specific API, providing additional JAXB properties that will allow fine-tuning the facility. Lastly, the specification proposes standard for XML Restrictions and Facets translation table to Bean Validation constraints and vice-versa.
1.6. **Major features**

- Validation is achieved by delegating to underlying Bean Validation implementation that must be present on classpath.
- Validation metadata is transferred during schema compilation and schema generation processes.
- Bean Validation annotations reflect XML Restrictions and Facets tied to the XML document representation of the object.
- Bean Validation annotations are automatically derived from the definitions provided in the XML schema and generated on the classes compiled during JAXB schema compilation processes.
- XML Restrictions and Facets are automatically created in XML Schema Descriptor for the marshalled object during Schema generation processes.

1.7. **Automatic Validation**

This specification defines the use of Bean Validation for automatic validation of JavaBeans upon the marshalling / unmarshalling validation events. These validation events occur prior to the object marshalling process and straight after XML document unmarshalling ends, but before the created object is published. If the validation events should find that any of the Validation constraints was not satisfied, the respective marshalling / unmarshalling process would fail and a BeanValidationException will be raised.

1.8. **Enabling Automatic Validation**

The **BEAN_VALIDATION_MODE** property determines whether the automatic lifecycle event validation will take place. Value of the property must be represented by enumeration **BeanValidationMode**:

- AUTO - If a Bean Validation provider is present in the environment, JAXB implementation provider performs automatic validation of JavaBeans. If no Bean Validation provider is present on classpath, validation will not be triggered.
- CALLBACK – The JAXB implementation provider must perform lifecycle event validation of JavaBeans. It is defined as an erroneous state if no Bean Validation provider is present on classpath.
- NONE – The JAXB implementation provider will not perform lifecycle event validation of JavaBeans.

**BEAN_VALIDATION_MODE** may be set during the creation of JAXBContext, Marshaller or Unmarshaller by providing **BeanValidationMode** property attribute. **Default BEAN_VALIDATION_MODE is AUTO.**

If AUTO validation mode is specified and a Bean Validation provider is present in the environment, the JAXB implementation provider must perform the automatic validation of
entities as described in section 1.7. If no Bean Validation provider is present in the environment, no validation takes place.

If CALLBACK validation mode is specified the JAXB implementation provider must perform the validation as described in section 1.7. It is an error if there is no Bean Validation provider present in the environment, and the provider must throw a BeanValidationException.

If NONE validation mode is specified by the BEAN_VALIDATION_MODE, then persistence provider must not perform lifecycle event validation.

1.9. Features of Automatic Validation

The requirement is that all objects that pass through the marshalling framework are valid. It is however not required that all objects that go through the marshalling framework must be validated. This slight difference allows for performance optimizations on level of this specification, i.e. only performing validation on objects that are constrained.

Validation will be performed by a Validator instance retrieved from the validator factory in use and invoking its validate method with the specified target groups. If the set of ConstraintViolation objects returned by the validate method is not empty, the JAXB Bean Validation provider must raise a BeanValidationException containing a copy of or reference to the returned set of ConstraintViolation objects.

Bean Validation specification allows for defining custom error messages on constraints via message argument. This feature is supported in the facility presented by this specification. Custom error message must be printed if validation does not pass and support for defining custom error messages must be added to XML Restrictions and Facets definitions via XML extensions.

This specification adds support for Bean Validation concept called Target Groups. A list of validation groups is targeted for validation. By default, the default Bean Validation group (the group Default) will be validated. This default validation behaviour can be overridden by specifying the target groups using the BEAN_VALIDATION_GROUPS property. This property may be set on JAXBContext, Marshaller or Unmarshaller. The value of this validation property must be a list of the targeted groups, represented by array of Class objects that represent the groups.

This specification dictates that both target groups and custom error message information can be set on XML Facets and Restrictions and must be transformed to Java BV constraints. This metadata must always be preserved during schema compilation and generation. The way of setting this custom metadata is standardized as follows:

<bv:facet type="assertFalse" message="false" groups="Group1"/>

Parameter type specifies the annotation’s class, parameter message specifies custom error message and parameter groups specifies custom target groups. If type is one of the annotations specified in the default package for Bean Validation annotations, only the short
name can be used, if the type is a custom annotation, fully qualified name has to be used. If more target groups were to be set, user should separate them with a comma and a space. Custom annotations are specified by XML Element defined in this specification. Its namespace is `xmlns: bv = "http://jaxb.dev.java.net/plugin/bean-validation"` and its definition is:

```
<xs:complexType>
  <xs:attribute name="message" type="xs:string"/>
  <xs:attribute name="groups" type="xs:string"/>
  <xs:attribute name="type" type="xs:string" use="required"/>
  <xs:attribute name="value" type="xs:string"/>
  <xs:annotation>
    <xs:documentation>
      Allows setting "value", which is available as a one-arg version of facet for user custom annotations.
    </xs:documentation>
  </xs:annotation>
</xs:complexType>
```

### 1.10. Providing the Validator Factory

In Java EE environments, a `ValidatorFactory` instance is made available by the Java EE container. The container is responsible for passing this validator factory to the JAXB implementation provider via the map that is passed as an argument to the `createContainerEntityManagerFactory` call. The mapping key used for specifying custom Validator Factory is `BEAN_VALIDATION_FACTORY`.

In Java SE environments, the application can pass the `ValidatorFactory` instance via the map that is passed as an argument to the JAXBContext, Marshaller or Unmarshaller as a property. The map key used must be the standard property name `org.eclipse.persistence.moxy.validation.factory`. If no `ValidatorFactory` instance is provided by the application, and if a Bean Validation provider is present on classpath, the BV in JAXB implementation provider must instantiate the `ValidatorFactory` using the default bootstrapping approach defined by the Bean Validation specification, namely `Validation.buildDefaultValidatorFactory()`.
1.11. Schema compilation

Bean Validation annotations will be automatically created for classes generated by XJC process. This will be achieved by plugging-in a BeanValidationPlugin that implements com.sun.tools.xjc.Plugin interface. To disable this behaviour user should disable this plugin. The mapping of Bean Validation constraints and XSD Restrictions and Facets is described in section 1.13.

1.12. Schema generation

By default, XML Restrictions and Facets will be created in XML Schema Descriptors, based on Bean Validation annotations present on the JavaBeans. To disable this behaviour, property GENERATE_FACETS can be used as an argument passed to JAXBContext factory methods. The property can be set to a value of Boolean.TRUE or Boolean.FALSE. Setting it to Boolean.FALSE will disable automatic Facets generation. Setting it to Boolean.TRUE would override previous disablements. The mapping of Bean Validation constraints and XSD Restrictions and Facets is described in section 1.13.

1.13. BV constraint to XML Restriction / Facet mapping table

The mappings are presented in Table 1.

**Table 1: Mapping table for BV constraints and XML Restrictions/Facets (Source: author).**

<table>
<thead>
<tr>
<th>BV Constraint</th>
<th>XML Restriction / Facet</th>
</tr>
</thead>
<tbody>
<tr>
<td>@AssertTrue</td>
<td></td>
</tr>
<tr>
<td>@AssertFalse</td>
<td></td>
</tr>
<tr>
<td>@DecimalMax</td>
<td>xs:maxExclusive, xs:maxInclusive</td>
</tr>
<tr>
<td>@DecimalMin</td>
<td>xs:minExclusive, xs:minInclusive</td>
</tr>
<tr>
<td>@Digits</td>
<td>xs:totalDigits, xs:fractionDigits</td>
</tr>
<tr>
<td>@Future</td>
<td></td>
</tr>
<tr>
<td>@Max</td>
<td>xs:maxExclusive, xs:maxInclusive</td>
</tr>
<tr>
<td>@Min</td>
<td>xs:minExclusive, xs:minInclusive</td>
</tr>
<tr>
<td>@NotNull</td>
<td>xs:nillable</td>
</tr>
<tr>
<td>@Null</td>
<td>xs:nillable</td>
</tr>
<tr>
<td>@Past</td>
<td></td>
</tr>
<tr>
<td>@Pattern</td>
<td>xs:pattern</td>
</tr>
<tr>
<td>@Size</td>
<td>xs:length, xs:maxLength, xs:minLength</td>
</tr>
</tbody>
</table>
Impedance too high to overcome is presented by some XML to Java mappings, as was case in JAXB 1.0 development. Some of the XML Facets do not naturally map to Java. The same obstruction was encountered with some Java BV restrictions, which do not map well to XML Schema. Those are @AssertTrue, @AssertFalse, @Future and @Past.

User is also allowed to define his custom constraints, which will be translated to Java annotations and vice-versa. Only one argument is supported for custom constraints, which is typical for XML Facets. All XML Facets are one argument facets, e.g. xs:nillable(true) is a one-argument facet and xs:minInclusive(15) is also one-argument facet. However, Bean Validation constraints can be multi-argument, e.g. @Size(min=x, max=y), which translates into two XML Facets: xs:minLength(x) and xs:maxLength(y).

---

1 “JAXB 2.0 will investigate support for all of W3C XML Schema including frequently requested features such as type and element substitution. To shorten the development cycle, JAXB 1.0 did not specify bindings for some of the W3C XML Schema features. These are listed in Appendix E.2 of the JAXB 1.0 specification. Some XML schema datatypes do not map naturally to Java™. Such additional Java™ datatypes have already been identified and communicated as requirements to the JAXP 1.3 (JSR 206) Expert Group. Any additional requirements will be communicated very early on in the JAXB 2.0 specification effort.” Viz https://jcp.org/en/jsr/detail?id=222
2. Implementation of “Bean Validation in JAXB” facility

2.1. Introduction

Author has created reference implementation for Bean Validation in JAXB specification. The implementation is in-built in MOXy, it is not separable, except for BeanValidationPlugin, which is a standalone Class pluggable into XJC. The implementation consists of four parts:

- API containing BV in JAXB properties,
- New Runtime classes and new callbacks to JAXB-Runtime part of MOXy,
- New Facets to Restrictions mapping classes and modifications to SchemaGen in MOXy,
- BeanValidationPlugin for XJC.

The goal was to enhance processes of the marshalling framework as depicted in Figure 1.

**Figure 1: Goal of Bean Validation in JAXB**

2.2. New marshalling framework with BV

New Runtime classes and new callbacks to JAXB-Runtime part of MOXy are illustrated in Figure 2.

**Figure 2: The marshalling framework with BV facility**
JAXBContext is the core class of all JAXB processes. It generates and stores object-to-xml mapping info, including metadata and class descriptors. Only one JAXBContext is needed per application, but more of them can be created in order to make the business logic more separated. Each JAXBContext can create one or more JAXBmarshallers and JAXBUnmarshallers. The function of these two classes is simple, to implement marshal, respectively unmarshal procedures. These three classes formed the base of the original marshaller framework.

JAXBMarshaller and JAXBUnmarshaller classes were enhanced with callback functions to the new JAXBBeanValidator. The callback methods are:

- shouldValidate – finds out if validation should take place
- validate – validates an object
- getConstraintViolations – retrieve a set of constraint violations from last validate call

JAXBBeanValidator could be without any exaggeration called the manager class of bean validation in JAXB. It is responsible for initializing Validators, for keeping track of the bean validation mode, deciding whether validation should take place based on the result of logical AND between boolean states: result{is validation mode allowing validation?} AND
result {is object constrained?}, and it also stores constraint violations from last validation call. The class uses automaton for its internal logic and was designed as a one-per-marshaller class, and as such is non thread-safe.

It delegates some of its logic to BeanValidationHelper, specifically the handling of logic for determining result {is object constrained?}. BeanValidationHelper is a singleton enumeration and it was designed as a thread-safe class, to allow concurrent access from multiple threads. Its main purpose is to optimize the performance of bean validation facility in JAXB. In fact, the “is object constrained?” check could be skipped altogether and just pass every marshalled / unmarshalled object to the underlying Bean Validation implementation for validation, which would not validate anything on non-constrained objects, and in the initial version the check was skipped. But then author raised concerns about the performance of the underlying bean validator (he was using reference implementation of Bean Validation for performance tests), and created a logic that decides whether an object should be passed to bean validator or if the validation should be skipped. This check scans for annotations on the object’s class. Fields and methods are searched for both default set of bean validation annotations and for custom constraints annotations. After this check, the object representing class descriptor of the marshalled object is stored in a cache and has a bool value associated with it (true if it is constrained, false if it is not constrained). On the next isConstrained call, the computation will be skipped and result retrieved from the cache. This technique is commonly known as memoization and is widely used in dynamic programming.

And that would be it, if Bean Validation specification had not allowed for configuration through external metadata. It states that constraints can be configured either through annotations or can be declared in „validation.xml“ configuration file. In order to achieve the goal of fully supporting Bean Validation 1.0 and 1.1 specifications, ValidationXMLReader class has been created. Its purpose is to parse the contents of „validation.xml“ file on classpath and search for constraints definition, if any are found, then store the constrained classes into the cache of singleton BeanValidationHelper. Its design is interesting as it reflects its unusual lifecycle. This class will be needed just for a very brief time period per application (per JVM) run. Parse validation.xml file once and store the results. Since the Bean Validation specification does not allow for change of external configuration during runtime, ValidationXMLReader class is not needed after it accomplishes its one-time task. Yet in order to achieve this task, it must create memory costly objects such as SAXParser and SAXParserFactory. What is worse, these objects cannot be bounded (declared in local scope of a method) and as such they impose a memory leak threat. So even if it feels natural to design classes that are in 1..* relation as singletons, it would be a suboptimal solution. The class was designed as an implementation of Callable interface that should be invoked only once per JAXBContext, perform, and then be marked for garbage collection with all its references to costly objects. Class scoped flags will prevent future creation of any instances of this class and thus effectively prevent memory leaks.
The enhanced process flow of JAXB in MOXy is portrayed on the next figure.

**Figure 3: Result - Enhanced JAXB process flow in MOXy**

The CPU resource savings achieved by the introduction of optimization „optional“ classes BeanValidationHelper and ValidationXMLReader are tremendous. As Figure 2 shows, the throughput of basic marshalling operations has been almost doubled, *i.e.* runs almost twice as fast.

**Figure 4: Optimization results**

<table>
<thead>
<tr>
<th>TestWorkOrderMarshal</th>
<th>thrpt</th>
<th>10</th>
<th>50</th>
<th>22452.601</th>
<th>75.6456</th>
<th>cps/s</th>
<th>(64.39 / 0.0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OutputStreamMarshal</td>
<td>thrpt</td>
<td>1</td>
<td>50</td>
<td>34685.3601</td>
<td>102.8361</td>
<td>cps/s</td>
<td>(82.82 / 0.0)</td>
</tr>
<tr>
<td>OrderResponseMarshal</td>
<td>thrpt</td>
<td>1</td>
<td>50</td>
<td>23666.7424</td>
<td>72.3233</td>
<td>cps/s</td>
<td>(57.02 / 0.0)</td>
</tr>
<tr>
<td>OutputStream.Marshal</td>
<td>thrpt</td>
<td>1</td>
<td>50</td>
<td>39340.0161</td>
<td>128.3776</td>
<td>cps/s</td>
<td>(92.08 / 0.0)</td>
</tr>
</tbody>
</table>

### 2.3. New SchemaGen with BV

The schema generation part of MOXy framework has been enhanced to automatically generate XSD Restrictions and Facets based on BV annotations found on Java classes. Since the schema generation process takes place upon creation of JAXBContext, the BV enhancement feature can be turned on/off by setting the following property (found in JAXBContextProperties) on JAXBContext:
Property for disabling/enabling generation of XML Facets during schemagen.

The mapped value must be of type Boolean.

If it’s true, then facets will be generated, based on the BV annotations.

If false, the BV annotations processing will be skipped during schemagen and no facets will be generated.

```java
public static final String GENERATE_FACETS = "eclipselink.generate.facets";
```

SchemaGen recognizes annotations provided by the BV API, including @Pattern.List.

If SchemaGen encounters a field annotated with both @NotNull and @XmlElement(nillable = true), it will raise the BeanValidationException.notNullAndNillable().

### 2.4. New Schema Compiler with BV

The XJC Plugin’s functionality is contained in a single new class org.eclipse.persistence.jaxb.plugins.BeanValidationPlugin.

XJC Plugin for generation of JSR349 (Bean Validation) annotations has two mods:

- “jsr303“ - enables backward compatibility.
- “simpleRegex” - disables translation of UNICODE XML regex into UNICODE Java regex.

Java ASCII regex are shorter to read.

The plugin is capable of generating the following annotations:

- @DecimalMax
- @DecimalMin
- @Digits
- @NotNull
- @Pattern
- @Size
- @Valid
- @AssertTrue
- @AssertFalse
- @Future
- @Past
And it reacts to the following XSD restrictions and facets, which it uses as the base for deriving the information for generation of aforementioned annotations:

- maxExclusive
- minExclusive
- maxInclusive
- minInclusive
- nillable
- pattern
- length
- maxLength
- minLength
- minOccurs
- maxOccurs

Basic usage for manual invocation of the plugin:

```
  xjc file.xsd -XBeanVal
```

Example usage with mods:

```
  xjc file.xsd -XBeanVal jsr303 simpleRegex
```

The plugin does support setting Target groups and Error message through binding customizations.

Example:

```
<xs:appinfo>
  <jxb:bindings
  node="/xs:schema/xs:complexType/xs:sequence/xs:element[@name='generic']">
    <bv:facet type="maxLength" message="Hello, world!" groups="Object"/>
    <bv:facet type="future" message="Welcome to the Future!"/>
  </jxb:bindings>
</xs:appinfo>
```

Supports custom-created BV annotations. Example:

```
<xs:appinfo>
  <jxb:bindings
  node="/xs:schema/xs:complexType/xs:sequence/xs:element[@name='generic']">
```

12
Running the Plugin

It can be run from command line:

```
compute -XBeanVal
```

, or by calling XJC's entry point, i.e. com.sun.tools.xjc.Driver class:

```
Driver.run(new String[] { schemaPath, “-extension”, “-XBeanVal” }, System.out,
System.out);
```

In MOXy, XJC is a tool which is not used by the runtime. But it is available to users via main() method on org.eclipse.persistence.jaxb.xjc.MOXyXJC class or the aforementioned Driver class.

JSR 303 option

If the jsr303 option is turned on, it downgrades the plugin to Bean Validation 1.0 API version. The only difference is handling the minExclusive and maxExclusive facets. Without this option, it sets “inclusive” parameter on DecimalMax and DecimalMin annotations to false, while with the option turned on it cannot set the attribute since it is not present in BV 1.0 API. Instead it will add 1 to the facet's value in the case of minExclusive or subtract 1 from the facet's value in case of maxExclusive.

Simple regex option

Turning this option on disables full compatibility with XML Schema regex. Only replaces \i \c \I \C regex shorthands if turned on. Else it does much more in order to ensure a full UNICODE compatibility between the XML regex shorthands and Java regex shorthands. It uses the RegexMutator inner classes for processing of the regex shorthands:
3. User guide

MOXy as JAXB provider

In order to take leverage of the new Bean Validation in JAXB specification, user needs to use MOXy, which is the reference implementation of the specification and currently also the only one, as his JAXB provider. MOXy can be used as JAXB provider in two ways: either the user must directly use classes EclipseLink specific JAXB implementation classes, or he can specify MOXy as his default JAXB implementation. The first approach is cumbersome and is not recommended, however, the usage is as follows:

org.eclipse.persistence.jaxb.JAXBContext context =
org.eclipse.persistence.JAXBContextFactory.createContext(...);

JAXBMarshaller marshaller = context.createMarshaller();

The cumbersome part is that fully qualified names have to be used and it is easy to omit those and make a mistake of using javax.xml.bind.JAXBContext.

The latter approach is the recommended one. To set MOXy as default JAXB provider, user has two options, either use an application server where MOXy is already the default JAXB implementation, such is the case for Oracle WebLogic\(^{2}\), and in other environments, leverage the META-INF/services mechanism to specify MOXy as the default JAXB provider:

1. Download EclipseLink version 2.6 or newer either from and add the downloaded eclipselink.jar on classpath of the application. At the moment, there is version 2.6.0-M3 of EclipseLink available from: sonatype\(^{3}\) and maven\(^{4}\) repositories. Official release version is scheduled for Q1-2015.

2. Create a JAR archive that contains a file called javax.xml.bind.JAXBContext in directory META-INF/services, which must be located in the sources directory of the application.

3. Set contents of the javax.xml.bind.JAXBContext file to a one-liner represented by the simple string of characters “org.eclipse.persistence.jaxb.JAXBContextFactory”

4. Add that jar to classpath of the application.

Then the jar will be discovered by the ServiceLoader mechanisms defined in Java™ SE platform and the user can use MOXy in an error-prone and simple way:

JAXBContext context = JAXBContext.newInstance(...);
Marshaller marshaller = context.createMarshaller();


\(^{3}\) [https://oss.sonatype.org/index.html#nexus-search;gav~org.eclipse.persistence~~~~](https://oss.sonatype.org/index.html#nexus-search;gav~org.eclipse.persistence~~~~)

where the package of the classes is the one defined in JAXB specification, i.e. javax.xml.bind.

User also has to ensure that a Bean Validation provider, such as Hibernate Validator (the BV reference implementation) is present on classpath. However, no further settings are needed, the provider will be located automatically.

**Using new Bean Validation in JAXB facility**

The usage of the new facility is simple, as it is turned on automatically in default settings:

```java
marshaller.marshal(validEmployee, file);
assert marshaller.getConstraintViolations().isEmpty();
```

If the user does not use any bean validation constraints, then he should use JAXB as usual. The second line is there to demonstrate that we can use the code in this way only if we know that the set of constraint violations is always empty.

Usage when the user knows that some JavaBeans are bean validation constrained:

```java
try {
    marshaller.marshal(invalidEmployee, file);
} catch (BeanValidationException e) {
    ... business logic handling invalid objects
}
```

If the employee object is invalid, i.e. its field values are violating some of the bean validation constraints, an exception will be raised. User should catch this exception and then handle this exceptional state, for example retrieve the set of violated constraints and log the information.

**Customizing the new facility**

Bean Validation in JAXB specification allows for various user settings and customizations. The basic one is to change the validation mode, e.g. turn it off. In order to turn the bean validation off on a particular marshaller instance, it can either be passed as an argument to the factory method or more conveniently can be set via a mutator:

```java
marshaller.setProperty(
    MarshallerProperties.BEAN_VALIDATION_MODE,
    BeanValidationMode.NONE );
```

To turn the validation off for the whole context, it is best to pass an argument to JAXBContextFactory factory method:

```java
Map<String, Object> props = new HashMap<>();
props.put(JAXBContextProperties.BEAN_VALIDATION_MODE, BeanValidationMode.NONE);

JAXBContext ctx = JAXBContext.newInstance(EMPLOYEE, props);

The following table lists all BV in JAXB properties:

**Table 2: List of all BV in JAXB properties (Source: author).**

<table>
<thead>
<tr>
<th>BEAN_VALIDATION_MODE</th>
<th>ON, AUTO, OFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEAN_VALIDATION_FACTORY</td>
<td>custom</td>
</tr>
<tr>
<td>BEAN_VALIDATION_GROUPS</td>
<td>custom</td>
</tr>
<tr>
<td>GENERATE_FACETS</td>
<td>boolean</td>
</tr>
<tr>
<td>BEAN_VALIDATION_NO_OPTIMISATION</td>
<td>boolean</td>
</tr>
</tbody>
</table>

and can be used in the same fashion as explained above for bean validation mode. Refer to JavaDoc for full details.
Programmer’s guide

Build

The source code for EclipseLink can be checked out from its official eclipse Git repository:

```
git clone
git://git.eclipse.org/gitroot/eclipselink/eclipselink.runtime.git
```

or manually from official repository website:

```
http://git.eclipse.org/c/eclipselink/eclipselink.runtime.git
```

To compile the source code, follow the following steps:

1. `git clone
git://git.eclipse.org/gitroot/eclipselink/eclipselink.runtime.git`
2. `cd eclipselink.runtime`
3. `mkdir ../extension.lib.external`
4. `<download junit+hamcrest-core, see https://github.com/junit-team/junit/wiki/Download-and-Install>`
5. `<unzip both jars and package the content into 'junit.jar'>
   >(or find junit.jar which bundles hamcrest and use that)>
6. `put both jars into the ../extension.lib.external folder`
7. `run command: ant -f antbuild.xml`

The source code will be compiled into a jar file in the same directory, called “eclipselink.jar”. JUnit and Hamrest libraries are required for the build and must be added manually, because they are not part of the project.

To run the tests, execute one of the following commands:

```
ant -f antbuild.xml test-moxy
ant -f antbuild.xml test-jaxb
ant -f antbuild.xml test-oxm
```

The first command runs all MOXy tests, including the ones for JAXB part and also the ones for SDO part. The latter two commands allow the programmer to run the tests separately.

For more information on building EclipseLink project, refer to the official Eclipse wiki:

```
https://wiki.eclipse.org/EclipseLink/Building
```

Contribute

The EclipseLink project is Open Source and contributions are welcome. It is recommended to sign up for the following mailing lists to participate in actual communication:

```
eclipselink-users@eclipse.org
```
eclipselink-dev@eclipse.org

There the programmer can consult with the EclipseLink community about bugs, desired features or anything else related to EclipseLink.

Another way to contribute is to participate on official forum, where the programmer can also file bugs:


Bugs can also be filed directly into the official Bugzilla database:

https://bugs.eclipse.org/bugs/

And lastly, it is recommended to follow the moxy and eclipselink tags on unofficial community forum StackOverflow:

http://stackoverflow.com/questions/tagged/moxy
http://stackoverflow.com/questions/tagged/eclipselink

Anyone can contribute new code. First a bug has to be filed in the official Bugzilla database describing the contribution. Then the proposed patch has to be attached to it. It will undergo review from EclipseLink committers. After the review, it will be committed by one of the EclipseLink committers and the original author will be noted as contributor. If a contributor should wish to become an EclipseLink committer, he should actively participate on forums and contribute regularly, after that he should refer to Committer Due Diligence Guidelines.

---

5 https://www.eclipse.org/legal/committerguidelines.php
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>JAXB</td>
<td>Java API for XML Binding</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible MarkUp Language</td>
</tr>
<tr>
<td>SE</td>
<td>Standard Edition</td>
</tr>
<tr>
<td>EE</td>
<td>Enterprise Edition</td>
</tr>
<tr>
<td>JDK</td>
<td>Java Development Toolkit</td>
</tr>
<tr>
<td>BV</td>
<td>Bean Validation</td>
</tr>
<tr>
<td>OXM</td>
<td>Object-to-XML Mapping</td>
</tr>
<tr>
<td>JRE</td>
<td>Java Runtime Environment</td>
</tr>
<tr>
<td>JSR</td>
<td>Java Specification Request</td>
</tr>
<tr>
<td>JCP</td>
<td>Java Community Process</td>
</tr>
</tbody>
</table>