

Hibernate JPA 2 Metamodel Generator

Reference Guide

1.2.0.Final

by Hardy Ferentschik

1. Introduction	1
1.1. What is it about?	1
1.2. Canonical Metamodel	2
2. Usage	5
2.1. Usage from the command line	5
2.1.1. Usage with Ant	5
2.1.2. Usage with Maven	6
2.2. Usage within the IDE	7
2.2.1. Idea	7
2.2.2. Eclipse	8
2.2.3. NetBeans	9
2.3. Processor specific options	9
A. Further information	11
References	13

Introduction

1.1. What is it about?

JPA 2 defines a new typesafe `Criteria` API which allows criteria queries to be constructed in a strongly-typed manner, using metamodel objects to provide type safety. For developers it is important that the task of the metamodel generation can be automated. Hibernate Static Metamodel Generator is an annotation processor based on the [[Pluggable Annotation Processing API](#)] with the task of creating JPA 2 static metamodel classes. The following example show two JPA 2 entities `Order` and `Item`, together with the metamodel class `Order_` and a typesafe query.

Example 1.1. JPA 2 annotated entities `Order` and `Item`

```
@Entity
public class Order {
    @Id
    @GeneratedValue
    Integer id;

    @ManyToOne
    Customer customer;

    @OneToMany
    Set<Item> items;
    BigDecimal totalCost;

    // standard setter/getter methods
    ...
}

@Entity
public class Item {
    @Id
    @GeneratedValue
    Integer id;

    int quantity;

    @ManyToOne
    Order order;

    // standard setter/getter methods
    ...
}
```

Example 1.2. Metamodel class `Order_`

```
@StaticMetamodel(Order.class)
```

```
public class Order_ {  
    public static volatile SingularAttribute<Order, Integer> id;  
    public static volatile SingularAttribute<Order, Customer> customer;  
    public static volatile SetAttribute<Order, Item> items;  
    public static volatile SingularAttribute<Order, BigDecimal> totalCost;  
}
```

Example 1.3. Typesafe criteria query

```
CriteriaBuilder cb = entityManager.getCriteriaBuilder();  
CriteriaQuery<Order> cq = cb.createQuery(Order.class);  
SetJoin<Order, Item> itemNode = cq.from(Order.class).join(Order_.items);  
cq.where( cb.equal(itemNode.get(Item_.id), 5 ) ).distinct(true);
```



Tip

The Metamodel Generator also takes into consideration xml configuration specified in orm.xml or mapping files specified in persistence.xml. However, if all configuration is in XML you need to add in at least on of the mapping file the following persistence unit metadata:

```
<persistence-unit-metadata>  
    <xml-mapping-metadata-complete/>  
</persistence-unit-metadata>
```

1.2. Canonical Metamodel

The structure of the metamodel classes is described in the [\[JPA 2 Specification\]](#), but for completeness the definition is repeated in the following paragraphs. Feel free to skip ahead to [Chapter 2, Usage](#) if you are not interested into the gory details.

The annotation processor produces for every managed class in the persistence unit a metamodel class based on these rules:

- For each managed class `x` in package `p`, a metamodel class `x_` in package `p` is created.
- The name of the metamodel class is derived from the name of the managed class by appending `"_"` to the name of the managed class.
- The metamodel class `x_` must be annotated with the `javax.persistence.StaticMetamodel` annotation.

- If class x extends another class s , where s is the most derived managed class (i.e., entity or mapped superclass) extended by x , then class x_* must extend class s_* , where s_* is the metamodel class created for s .
- For every persistent non-collection-valued attribute y declared by class x , where the type of y is Y , the metamodel class must contain a declaration as follows:

```
public static volatile SingularAttribute<X, Y> y;
```

- For every persistent collection-valued attribute z declared by class x , where the element type of z is Z , the metamodel class must contain a declaration as follows:
 - if the collection type of z is `java.util.Collection`, then

```
public static volatile CollectionAttribute<X, Z> z;
```

- if the collection type of z is `java.util.Set`, then

```
public static volatile SetAttribute<X, Z> z;
```

- if the collection type of z is `java.util.List`, then

```
public static volatile ListAttribute<X, Z> z;
```

- if the collection type of z is `java.util.Map`, then

```
public static volatile MapAttribute<X, K, Z> z;
```

where K is the type of the key of the map in class X

Import statements must be included for the needed `javax.persistence.metamodel` types as appropriate and all classes x , Y , Z , and K .

Usage

The jar file for the annotation processor can be found in the [JBoss Maven repository](http://repository.jboss.com/) [http://repository.jboss.com/] using [Example 2.1, “Maven dependency”](#).

Example 2.1. Maven dependency

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-jpamodelgen</artifactId>
  <version>1.0.0</version>
</dependency>
```

Alternatively, a full distribution package can be downloaded from [SourceForge](http://sourceforge.net/projects/hibernate/files/hibernate-jpamodelgen) [http://sourceforge.net/projects/hibernate/files/hibernate-jpamodelgen].

In most cases the annotation processor will automatically run provided the processor jar is added to the classpath and a JDK 6 is used. This happens due to [Java's Service Provider](http://java.sun.com/j2se/1.4.2/docs/guide/jar/jar.html#Service%20Provider) [http://java.sun.com/j2se/1.4.2/docs/guide/jar/jar.html#Service%20Provider] contract and the fact the the Hibernate Static Metamodel Generator jar files contains the file `javax.annotation.processing.Processor` in the `META-INF/services` directory. The fully qualified name of the processor itself is: `org.hibernate.jpamodelgen.JPAMetaModelEntityProcessor`.



Note

The use of a Java 6 compiler is a prerequisite.

2.1. Usage from the command line

2.1.1. Usage with Ant

As mentioned before, the annotation processor will run automatically each time the Java compiler is called, provided the jar file is on the classpath. Sometimes, however, it is useful to control the annotation processing in more detail, for example if you exclusively want to run the processor without compiling any other source files. [Example 2.2, “Javac Task configuration”](#) shows how Ant's [Javac Task](http://ant.apache.org/manual/CoreTasks/javac.html) [http://ant.apache.org/manual/CoreTasks/javac.html] can be configured to just run annotation processing.

Example 2.2. Javac Task configuration

```
<javac srcdir="${src.dir}"
      destdir="${target.dir}"
```

```
failonerror="false"
fork="true"
classpath="${classpath}">
  <compilerarg value="-proc:only"/>
</javac>
```

The option *-proc:only* instructs the compiler to just run the annotation processing. You can also completely disable processing by specifying *-proc:none*.



Tip

Run 'javac -help' to see which other annotation processor relevant options can be specified.

2.1.2. Usage with Maven

There are several ways of running the annotation processor as part of a Maven build. Again, it will automatically run if you are using a JDK 6 compiler and the annotation processor jar is on the classpath. In case you have more than one annotation processors on your classpath you can explicitly pass the processor option to the compiler plugin:

Example 2.3. Maven compiler plugin configuration - direct execution

```
<plugin>
  <artifactId>maven-compiler-plugin</artifactId>
  <configuration>
    <source>1.6</source>
    <target>1.6</target>
    <compilerArguments>
      <processor>org.hibernate.jpamodelgen.JPAMetaModelEntityProcessor</processor>
    </compilerArguments>
  </configuration>
</plugin>
```

The maven-compiler-plugin approach has the disadvantage that the maven compiler plugin does currently not allow to specify multiple compiler arguments ([MCOMPILER-62](http://jira.codehaus.org/browse/MCOMPILER-62) [http://jira.codehaus.org/browse/MCOMPILER-62]) and that messages from the Messenger API are suppressed ([MCOMPILER-66](http://jira.codehaus.org/browse/MCOMPILER-66) [http://jira.codehaus.org/browse/MCOMPILER-66]). A better approach is to disable annotation processing for the compiler plugin as seen in [Example 2.4](#), "[Maven compiler plugin configuration - indirect execution](#)".

Example 2.4. Maven compiler plugin configuration - indirect execution

```
<plugin>
  <artifactId>maven-compiler-plugin</artifactId>
  <configuration>
```

```

    <source>1.6</source>
    <target>1.6</target>
    <compilerArgument>-proc:none</compilerArgument>
  </configuration>
</plugin>

```

Once disabled, the [maven-processor-plugin](http://code.google.com/p/maven-annotation-plugin/) [http://code.google.com/p/maven-annotation-plugin/] for annotation processing can be used. The configuration can be seen in [Example 2.5](#), “Configuration with maven-processor-plugin”.

Example 2.5. Configuration with maven-processor-plugin

```

<plugin>
  <groupId>org.bsc.maven</groupId>
  <artifactId>maven-processor-plugin</artifactId>
  <version>2.0.5</version>
  <executions>
    <execution>
      <id>process</id>
      <goals>
        <goal>process</goal>
      </goals>
      <phase>generate-sources</phase>
      <configuration>
        <processors>
          <processor>org.hibernate.jpamodelgen.JPAMetaModelEntityProcessor</processor>
        </processors>
      </configuration>
    </execution>
  </executions>
  <dependencies>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-jpamodelgen</artifactId>
      <version>1.2.0.Final</version>
    </dependency>
  </dependencies>
</plugin>

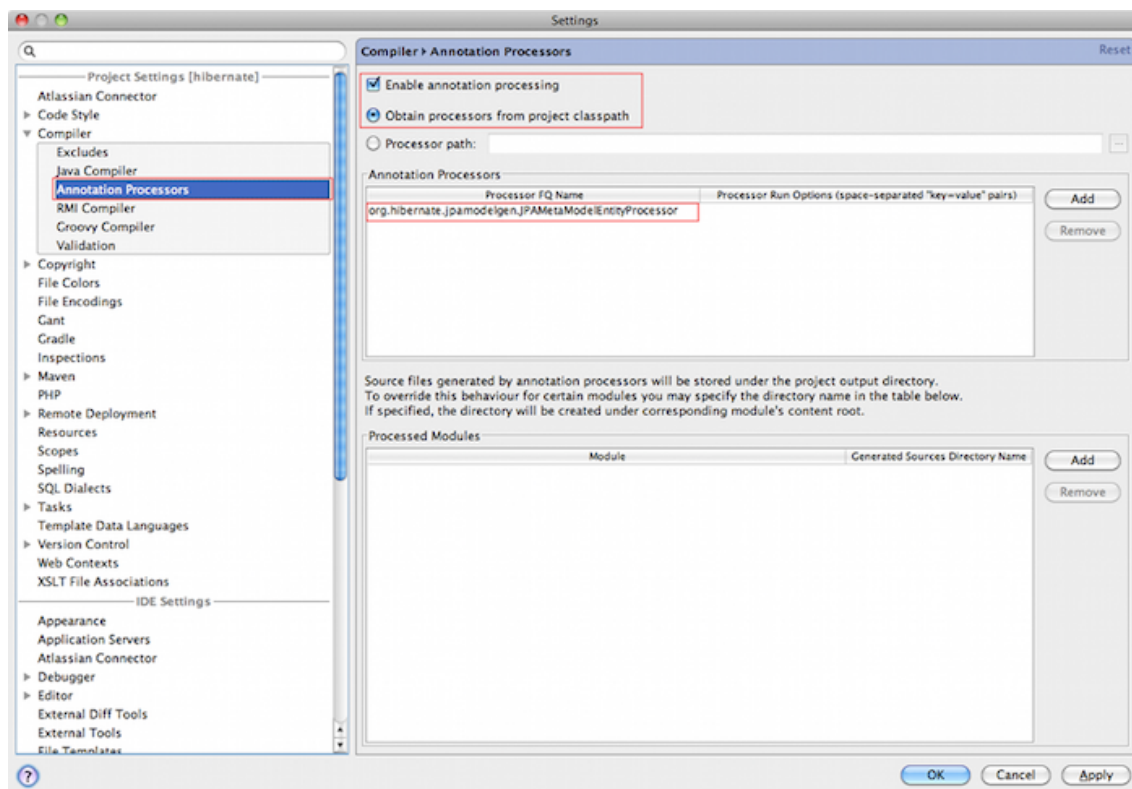
```

2.2. Usage within the IDE

Of course you also want to have annotation processing available in your favorite IDE. The following paragraphs and screenshots show you how to enable the Hibernate Static Metamodel Generator within your IDE.

2.2.1. Idea

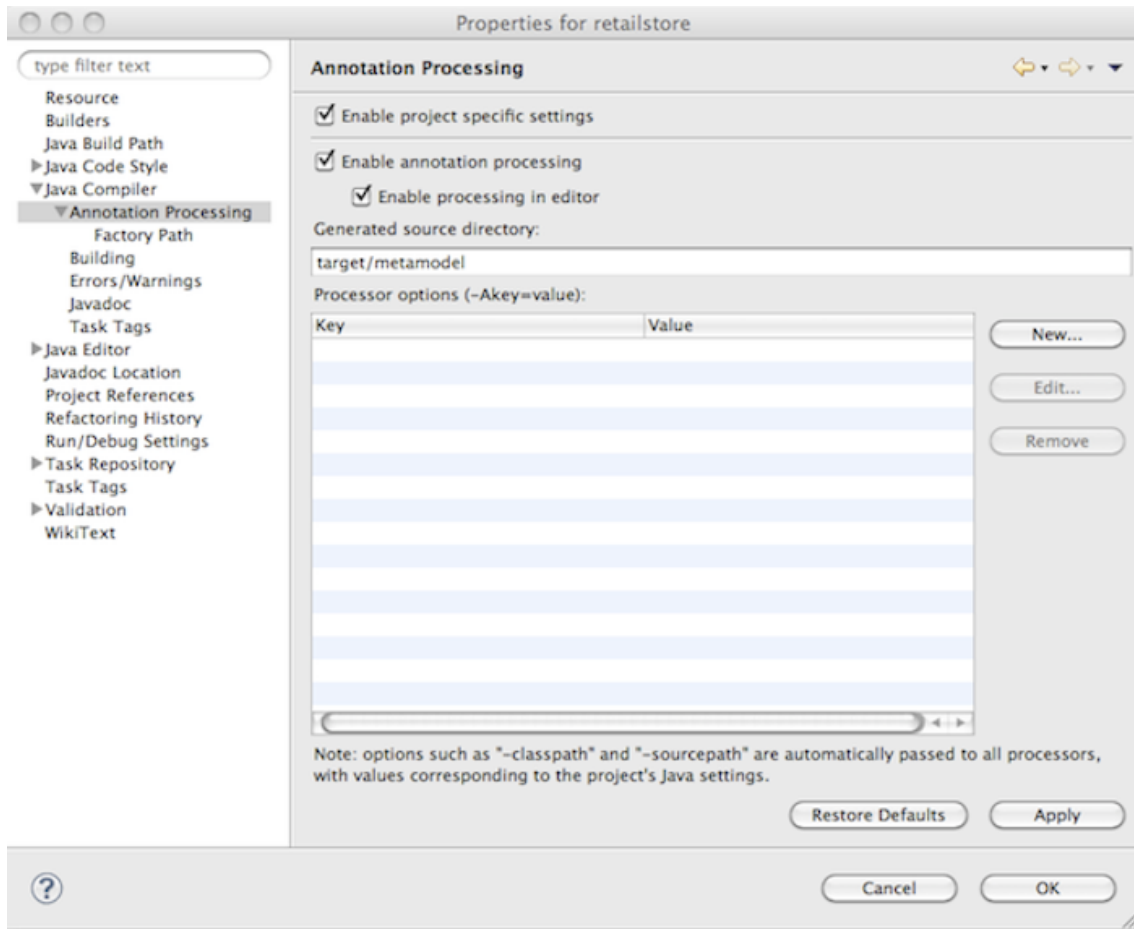
IntelliJ Idea contains from version 9.x onwards a specific configuration section for annotation processing under the project settings window. The screenshots show you how to configure the Hibernate Static Metamodel Generator.



In the annotation processor configuration, enable annotation processing and select obtain from project classpath. Add the annotation processor name `org.hibernate.jpamodelgen.JPAMetaModelEntityProcessor` (and optionally the annotation processor options). Select the module(s) containing your entities. If you have configured Maven as recommended, it is best to select the same output directory for the generated classes. At the time of writing, it is `target/generated-sources/apt`. That way, the generated classes will be available in IntelliJ Idea.

2.2.2. Eclipse

In Eclipse, from the Galileo release onwards, exists an additional configuration section under Java Compiler. There you can configure all kinds of aspects of annotation processing. Just check the "Enable annotation processing" option, configure the directory for the generated sources and finally add the Hibernate Static Metamodel Generator and JPA 2 jar files to the factory path.



2.2.3. NetBeans

Netbeans support for annotation processors is at the time of this writing still in the making. Refer to NetBeans issues [111065](http://www.netbeans.org/issues/show_bug.cgi?id=111065) [http://www.netbeans.org/issues/show_bug.cgi?id=111065], [111293](http://www.netbeans.org/issues/show_bug.cgi?id=111293) [http://www.netbeans.org/issues/show_bug.cgi?id=111293] and [111294](http://www.netbeans.org/issues/show_bug.cgi?id=111294) [http://www.netbeans.org/issues/show_bug.cgi?id=111294].

2.3. Processor specific options

The Hibernate Static Metamodel Generator accepts a series of custom options which can be passed to the processor in the format `-A[property]=[value]`. The supported properties are:

Table 2.1. Annotation processor options (passed via `-A[property]=[value]`)

Option name	Option value and usage
debug	If set to <code>true</code> additional trace information will be outputted by the processor
persistenceXml	Per default the processor looks in <code>/META-INF</code> for <code>persistence.xml</code> . Specifying this option a

	<code>persistence.xml</code> file from a different location can be specified (has to be on the classpath)
<code>ormXml</code>	Allows to specify additional entity mapping files. The specified value for this option is a comma separated string of mapping file names. Even when this option is specified / <code>META-INF/orm.xml</code> is implicit.
<code>lazyXmlParsing</code>	Possible values are <code>true</code> or <code>false</code> . If set to <code>true</code> the annotation processor tries to determine whether any of the xml files has changed between invocations and if unchanged skips the xml parsing. This feature is experimental and contains the risk of wrong results in some cases of mixed mode configurations. To determine whether a file has been modified a temporary file <code>Hibernate-Static-Metamodel-Generator.tmp</code> is used. This file gets created in the <code>java.io.tmpdir</code> directory.
<code>fullyAnnotationConfigured</code>	If set to <code>true</code> the processor will ignore <code>orm.xml</code> and <code>persistence.xml</code> .
<code>addGeneratedAnnotation</code>	If set to <code>true</code> the processor will add the <code>@Generated</code> to the generated Java source file. Adding this annotation using JDK 5 will cause a compilation error. In this case set the flag to <code>false</code> . The default for this option is <code>true</code>
<code>addGenerationDate</code>	If set to <code>true</code> the generation date of the metamodel class will be inserted in the date parameter of the <code>@Generated</code> annotation. The default is <code>false</code> . This parameter is ignored if <code>addGeneratedAnnotation</code> is set to <code>false</code> .
<code>addSuppressWarningsAnnotation</code>	If set to <code>true</code> the processor will add <code>@SuppressWarnings("all")</code> to the generated Java source file. Per default this annotation is not generated. See also METAGEN-50 [https://hibernate.onjira.com/browse/METAGEN-50].

Appendix A. Further information

For further usage question or problems consult the [Hibernate Forum](https://forum.hibernate.org/viewforum.php?f=9) [https://forum.hibernate.org/viewforum.php?f=9]. For bug reports use the [METAGEN](http://opensource.atlassian.com/projects/hibernate/browse/METAGEN) [http://opensource.atlassian.com/projects/hibernate/browse/METAGEN] project in the Hibernate Jira instance. Feedback is always welcome.

References

[Pluggable Annotation Processing API] *JSR 269: Pluggable Annotation Processing API*. Copyright © 2006 SUN MICROSYSTEMS, INC.. <jsr-269-feedback@sun.com> [JSR 269 JCP Page](http://jcp.org/en/jsr/detail?id=269) [http://jcp.org/en/jsr/detail?id=269].

[JPA 2 Specification] *JSR 317: Java™ Persistence API, Version 2.0*. Java Persistence 2.0 Expert Group. . Copyright © 2009 SUN MICROSYSTEMS, INC.. <jsr-317-feedback@sun.com> [JSR 317 JCP Page](http://jcp.org/en/jsr/detail?id=317) [http://jcp.org/en/jsr/detail?id=317].

