The Hibernate/JPA POJO Generator

When you already have a database and you want to use JPA or Hibernate to access that database you need something that will generate those POJO classes for you with the appropriate annotations. There are some existing solutions but I was not happy with them, so this is Yet Another Pojo Generator.

The generator gets generates as part of the build of DomUI and can be found as utilities/hibernate-generator/target/hibernate-generator.jar, and can be executed as:

java -jar hibernate-generator.jar

The generator so far should work for PostgreSQL and Oracle, with PostgreSQL the one that has been tested. Adding new database types should not be too hard.

The generator does the following:

- It generates a new POJO class for every table in the schema(s) specified
- If a POJO class already exists for the table *it edits the POJO file and updates it with the database definition*:
  - New columns are added to the existing POJO
  - Removed columns are, well, removed
  - Metadata is updated where applicable
- Existing code in POJO classes is mostly left alone, so it should be reasonably safe to re-run the generator on already existing code.
- It has basic support for compound primary keys; it should properly generate XxxxId classes for class Xxxx if that table as a compound primary key
- It adds all relations that are properly defined by foreign keys
  - It adds @ManyToOne parent references in the child class
  - It adds @OneToMany child list references as List<T> in the parent, with T the type of the child class
  - This however will not work for compound PK's.
- It automatically recognizes *and uses* any @MappedSuperClass class as a base class for those tables that share colums with that base class.
- It generates/updates [classname].properties files with the new properties in the class, so that these files can be used as resource bundles for the property names of that class.
- It generates a class HibernateConfigurator which contains methods that add all classes to Hibernate's config.

Usage

An example command line invocation would be:

```
java -jar hibernate-generator.jar -dbtype postgres -db dbusername:dbpasswordl@localhost/database_name -pkgroot
org.mydomain.myprogram.database -source /home/jal/myproject/src/main/java -s pdi_meta -s source_mapping -s auth
-s definition -s sectormodel
```

This connects to a PostgreSQL database with the specified username and passwords, loads all tables from the schema's pdi_meta, source_mapping, auth and sectormodel, then generates/updates all classes into the specified directory and package.

The directory and package are *separate*, so the final directory for the classes is formed by adding the source AND the package name.

The options supported are:

Option	Short
-db	Database connection string as username:password@hostname[:port]/databasename (required)
-dbtype	The database type: postgres or oracle (required)
-source	Specifies the root of the source directory for both existing and generated sources
-pkg	The package name for the generated classes
-destroy-constructors	When set, this destroys all constructors in existing Java classes. It can be used to get rid of the silliness generated by the hibernate pojo tool
-ffr	When set all field names are forcefully renamed to the name as decided by column name and prefix.
-frm	When set all getter and setter methods in existing classes are renamed to whatever the property name is calculated to be
-nb, -no-bundles	Disable generation of .properties bundles
-no-baseclass	Do not try to find base classes for new pojo's
-no-deserial	Postgres 'serial' columns are actually just columns with a 'default' which retrieves a value from a generated sequence. By default the code will find this sequence and generate a SEQUENCE type of ID generator. Setting this option will cause the code to use the generated. IDENTITY method.

-no-onechar-boolean	By default, all columns found that are (var)char with a size of 1 and that contain <= 2 distinct values are generated as boolean with an appropriate @Type annotation. This option disables that.
-noi, -no-identifyable	By default all generated classes will implement IIdentifyable<T>. This disables generating that.
-s, -schema	Adds a schema to reverse engineer

Generated code

When the generator writes its output it will always make a backup copy of all files it overwrites, as ".old" files. In addition, when a .old file exists, the generator will always read *that* file as the source file when called again. This allows you to re-run the generator many times until the result is as desired; it will never re-read the code it has generated but instead the data that was present at the start.

Once you are happy with the result you should remove the .old files.

A .old file 0 bytes long is generated for all new files, so that the generator knows that those are not original either. Keep that in mind if you want to rename the .old files back.

Specifying exceptions and overriding names

By default the generator will try a bit to concoct reasonable names for classes and properties, but the quality is very dependent on whatever is used in the database. If you do not like the generated names you have two choices:

- Use your favorite IDE to rename the field, getter and setter. This will work fine, because the next time the generator is run it will know that new name from the .java source and use that by default.
- Add an override to the GenHib.xml file

The GenHib.xml file gets generated/updated by the generator every time it is run. After the first run it will contain all possible options that can be set for all tables and columns with an "empty" value which means "use the default, Luke". You can simply edit this file to override the names and other things for classes. So a typical run would be:

- Run the generator to create all classes
- Inspect the classes, and change everything you hate by updating GenHib.xml
- Repeat from step one until satisfied

Do not forget to add the .xml file to your VCS.