Jrsd v. 2.8 manual

1 (Long) Introduction

Would you write a java program like this ? :

```
<java-source-program>
  <java-class-file name="FirstApplet.java">
    <import module="java.applet.*"/>
    <import module="java.awt.*"/>
    <class name="FirstApplet" visibility="public"
           line="5" col="0" end-line="11" end-col="0"
comment="// do not forget this import statement!// Or this one for the graphics!">
    <superclass name="Applet"/>
      <method name="paint" visibility="public"
              id="FirstApplet:mth-15" line="8" col="2" end-line="10" end-col="2"
comment="// this method displays the applet.// the Graphics class is how you do all
the drawing in Java">
        <type name="void" primitive="true"/>
        <formal-arguments>
          <formal-argument name="g" id="FirstApplet:frm-13">
            <type name="Graphics"/>
          </formal-argument>
        </formal-arguments>
        <block line="8" col="32" end-line="10" end-col="2" comment="// do not forget
this import statement!// Or this one for the graphics!// this method displays the
applet.// the Graphics class is how you do all the drawing in Java">
          <send message="drawString">
            <target>
              <var-ref name="g" idref="FirstApplet:frm-13"/>
            </target>
            <arguments>
              <literal-string value="FirstApplet"/>
              <literal-number kind="integer" value="25"/>
              <literal-number kind="integer" value="50"/>
            </arguments>
          </send>
        </block>
      </method>
    </class>
  </java-class-file>
</java-source-program>
```

instead of :

```
import java.applet.*;   // do not forget this import statement!
import java.awt.*;      // Or this one for the graphics!

public class FirstApplet extends Applet {
  // this method displays the applet.
  // the Graphics class is how you do all the drawing in Java
  public void paint(Graphics g) {
    g.drawString("FirstApplet", 25, 50);
  }
}
```

?

(this example was “hijacked” from javaML, a project whose purpose is to make a java representation more suited for automated, machine processing)

Writing mappings for various ORMs just feels like this for me. Wouldn't it be nice to start with the SQL's data definition of the table, and to add mapping clauses afterwards, side by side with the SQL ? This is the goal of jrsd : you start with the SQL ddl statements, and you add constructs to indicate how SimpleORM should map

this to java code. Jrsd stands for “Java Relational Simpleorm Descriptors”, because the statements you write are “descriptors” that glue java and the relational database part, using the well-thought simpleorm API. The jrsd precompiler will generates the SQL files *and* the java files for you, triggered by ant, or directly, by calling jrsd on the command line. Simpleorm can also generate SQL from its declarations, but you don't see the SQL code except when it has been executed, which can be annoying, for example if you work with database administrators who review your SQL before *they* execute it at the production database (actually simpleorm can generated some SQL now using methods like `SDriver.createTableSQL()` for example, but it is still cumbersome to control the SQL in my opinion). Jrsd gives you lots of places to add your own SQL code, and thus to better control the generated SQL. The generated SQL is just a bunch of files, so this is easy to send and review. This is more in the “*don't be afraid of SQL*” approach, that we like in simpleorm. The goal is the same that simpleorm : one place to make modifications. Of course if you use jrsd for SQL generation, don't use simpleorm's DDL methods !

This project is complementary to Simpleorm, but is not meant to be included inside the simpleorm project, because of a few differences in the approach :

- In simpleorm, columns are declared inside the java code; jrsd keeps separated files that generate the java code. Jrsd should be used only for people that prefer to maintain a separate file, this is not the case for everyone.
- The simpleorm library is small, and this small size is an argument for its adoption in many cases. Jrsd adds another 80 K for an extra comfort not needed by everyone. The approach in jrsd is to have *simple*orm, not necessarily *small*orm.
- Jrsd is not as well-tested as simpleorm, it is still beta code (although under Oracle I have now more than 3 years of experience).
- Jrsd adds extra utility classes to make some particular uses more convenient (use of BLOBs/CLOBs, value transfer objects, ...). This is a bit of a “code bloat” and is not in the philosophy of simpleorm.
- Jrsd uses log4j, which adds another 350 K of jars to distribute.

2 Why use jrsd + simpleorm ?

The advantages in using jrsd are :

- You start with an SQL file (well, sort of); not an XML file, not a java file. For an existing table, you can start from a reverse-engineered sql file, this is quite fast. Also the jrsd files are quite compact and self-documenting, they are more convenient to read than the java files.
- Automatic generation of SQL files that are like hand-made SQL files (hand-written parts can be added at most places)
- Automatic generation of the java classes for the orm, with all the wanted getters and setters; this adds more comfort when using IDEs (Eclipse, Netbeans, …) that have code completion feature, and makes the business code that uses those classes more concise and clear. Also lots of libraries ask for

this (struts 2, template libraries, …).

- Transport mechanism which simplifies transfers between a javabean and a simpleorm record. The use of plain old java objects removes the dependence to simpleorm's datasets, which adds more flexibility. Datasets are more safe, but this safety has a price. Transport objects makes the code simpler to write for the easy cases.
- All parts are optional, simpleorm and/or direct jdbc can always be used when wanted, so using jrsd is not blocking. Jrsd just gives alternatives, it is of course not meant as a replacement for simpleorm methods. The user always has the choice, jrsd just gives more to choose from.
- Small utility classes that add more capabilities to simpleorm/jdbc : sql execution from resource files, value objects as an alternative to the SrecordInstance, classes to help with oracle, with blobs, with templating frameworks, with log4j logging, ...
- SQL file version/checksum mechanism, that prevents an accidental change in the database model. After 24 hours, to change something in the database (a column name, type, …) you must either delete the old sql file manually, or declare a new version of the table. This avoids accidental unwanted changes, because databases are not as easily changed (and safe to change !) as codebases.
- All the extra elements are there to ultimately **make your business code more readable**. You write extra code outside the business methods (factory methods inside your SrecordInstance extension classes, DbOps methods, transport object classes etc.) so that your business code gets very simple to read, follow and debug.

2.1 History of jrsd

When I started to use simpleorm around 2005, I was not very satisfied with the SQL code generation of simpleorm (and our database administrator even less...). I started to write the SQL separately, and then the orm classes. After a while, I used a system with XML files and stylesheets. This automated things much farther, but the whole thing was not very readable. After getting tired of XML file maintenance, I realized that the relevant part was in the SQL files, and played with a small ANTLR grammar. This worked really well, and I added features and constructs as soon as I needed them in a project. When simpleorm was released with the improved dataset model, I had to rewrite much of the code and the grammar. Now after some years of use, I am reasonably satisfied with the tool, and I thought it was time to give something back to the simpleorm community who brought us this very useful orm tool and library. The name “jrsd” is not very nice nor self-speaking, but it has one advantage : is is uniquely referenced by web search engines.

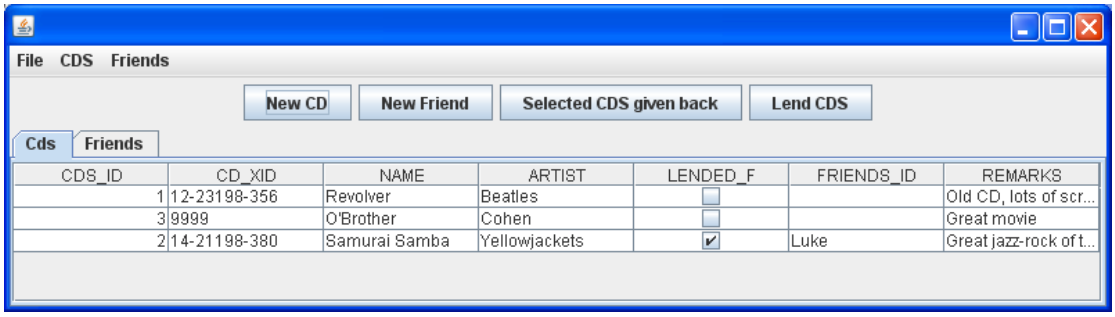
3 For the impatient

Go to the directory where you have unpacked jrsd, and type

```
ant demo
```

This will build a demo in “dist-demo”, and launch a simple demo application, that uses the h2 java database, will be run. The application lets you describe your audio CDs, and keeps track of the CDs you have lended to your friends. This is just for demonstration purposes, you should not use this application for real.

Here is a screenshot of the “Lending CDs” application :



You can have a look at the “`dist-demo\jrsd`” dir for the jrsd files, and at the “`dist-demo\src\sql`” dir for the generated sql files. The generated simpleorm java source files have been generated in the “`dist-demo\src\org\hmn\jrsd\demo\orm`” dir.

For a quick overview of what is generated and how, I have put a simple descriptor for a PATIENT table. A patient has a unique identifier assigned by an hospital, and the usual well known fields for first name, last name, birth date. Other fields (sex, usual name, etc.) have been left out for clarity. Two files will be generated for the `dist-demo\jrsd\PATIENT.java` :

- `dist-demo\src\org\hmn\jrsd\demo\orm\PATIENT.java`
- `dist-demo\src\sql\table-PATIENT_v1_2392001264.sql`

We will start with these files to get a quick glance of what is declared and generated.

4 A simple .jrsd file

The file **`PATIENT.jrsd`** found in **`demo/jrsd/PATIENT.jrsd`** :

```
jrsd 2;

/* PATIENT */
table PATIENT version 1 :
  simpleorm "org.hmn.jrsd.demo.orm.PATIENT" as s
(
  /* Unique id of the patient */
  PID varchar(16) : s;
  /* Last name */
  LAST_NAME VARCHAR(64) : s;
  /* First name */
  FIRST_NAME VARCHAR(64) : s;
  /* Date of birth */
  BIRTH_DATE TIMESTAMP : s;
  primary key (PID);
)
;
```

5 The generated .java and .sql files

The generated SQL file, **`table-PATIENT_v1_2392001264.sql`** :

```
/* PATIENT */
CREATE TABLE PATIENT    /* version 1 */ (
  /* Unique id of the patient */
  PID varchar(16),
  /* Last name */
  LAST_NAME varchar(64),
  /* First name */
  FIRST_NAME varchar(64),
  /* Date of birth */
  BIRTH_DATE timestamp,
  PRIMARY KEY (PID)
)
;
```

The generated java file, **`org/hmn/jrsd/demo/orm/PATIENT.java`** :

```
/* simpleorm class org.hmn.jrsd.demo.orm.PATIENT */
/*****************************************************************************
* WARNING - This class was automatically generated by the jrsd tool. Any      *
* modifications made to this source will be overwritten the next time that    *
* jrsd is run. Modify the .jrsd file instead.                                 *
*****************************************************************************/
package org.hmn.jrsd.demo.orm;
import java.math.BigDecimal;
import java.sql.Timestamp;
import java.sql.Time;
import java.sql.Blob;
import simpleorm.dataset.*; //because this file is generated, we can not know in
advance what is used or not

import org.hmn.jrsd.rt.TransportFieldMapping;
import org.hmn.jrsd.rt.TransportMappings;
import org.hmn.jrsd.rt.FieldList;


public class PATIENT
extends SRecordInstance
{
  /** indicates the jrsd table version number */
  public int getSqlSourceVersion() { return 1; }

  public static final SRecordMeta<org.hmn.jrsd.demo.orm.PATIENT> meta =
      new
SRecordMeta<org.hmn.jrsd.demo.orm.PATIENT>(org.hmn.jrsd.demo.orm.PATIENT.class,
"PATIENT")
      ;


  /**  Unique id of the patient  */
  static final public SFieldString PID =
    new SFieldString(meta, "PID"     , 16    , SFieldFlags.SPRIMARY_KEY)

    ;
  /**  Last name  */
  static final public SFieldString LAST_NAME =
    new SFieldString(meta, "LAST_NAME"     , 64  )

    ;
  /**  First name  */
  static final public SFieldString FIRST_NAME =
    new SFieldString(meta, "FIRST_NAME"     , 64  )

    ;
  /**  Date of birth  */
  static final public SFieldTimestamp BIRTH_DATE =
    new SFieldTimestamp(meta, "BIRTH_DATE"   )

    ;



  public String getPID() {
    return (String)getString(PID);
```

```
  }

  public void setPID(String v) {
    setString(PID, v);
  }

  public String getLAST_NAME() {
    return (String)getString(LAST_NAME);
  }

  public void setLAST_NAME(String v) {
    setString(LAST_NAME, v);
  }

  public String getFIRST_NAME() {
    return (String)getString(FIRST_NAME);
  }

  public void setFIRST_NAME(String v) {
    setString(FIRST_NAME, v);
  }

  public Timestamp getBIRTH_DATE() {
    return (Timestamp)getTimestamp(BIRTH_DATE);
  }

  public void setBIRTH_DATE(Timestamp v) {
    setTimestamp(BIRTH_DATE, v);
  }

  @Override
  public SRecordMeta<org.hmn.jrsd.demo.orm.PATIENT> getMeta() { return meta; }



}
```

The command that was used for the generation :

bin/jrsd -srcroot src -sqlroot sql jrsd/PATIENT.jrsd

6 Format of a .jrsd file

6.1 Preamble

Every jrsd file starts with the string “jrsd” followed by a version number. Currently the version number to use is “2”.

6.2 Statements and semicolons

Most jrsd statements end with a semicolon. This may be confusing at first because in SQL column declarations are separated with commas. Commas are used to separate multiple statements in the mapping part.

6.3 Sql / Java duality

Most declarations have an sql part and a java part. The sql part comes first, the java part is optional and comes afterwards. The java and sql parts are separated by a colon (“:”). SQL Tables are mapped to simpleorm SrecordInstance(s), and SQL Columns

are mapped to SFields of record instances. If an SQL element is not mapped to a java element, it will only appear in the SQL. In the “java” part, mustiple statements can occur, they must be separated by commas.

6.4 Aliases

Most elements are referenced by an alias; when you declare them, you must add “as <alias_name>”, this alias is then used in further declarations.

Example :

```
simpleorm "org.hmn.jrsd.demo.orm.PATIENT" as s
```

The alias “s” will be used to reference the simpleorm SrecordInstance class PATIENT

6.5 Default names

When no name is given in the java declaration part, the name is generated using the column name. This is usually sufficient, and makes the java part very small (just the alias of the simpleorm class most of the time). Generated names will have the same case than the column name; this can generate warnings in java but otherwise it is mostly harmless. Ex :

```
  PID varchar(16) : s;
```

Generated field :

```
  static final public SFieldString PID =
    new SFieldString(meta, "PID"     , 16    , SfieldFlags.SPRIMARY_KEY)
```

You can force the case of the generated fields with some options (see below) if this is really necessary.

6.6 Explicit names

If the names in the table and the record don't match (not recommended), you can declare the name explicitly by just using a dot and the identifier. Ex :

```
  PID varchar(16) : s.patientIdentifier;
```

Generated field :

```
  static final public SFieldString patientIdentifier =
    new SFieldString(meta, "PID"     , 16    , SfieldFlags.SPRIMARY_KEY)
```

6.7 Table declaration

The format of the table declaration is :

```
table <tablename> version nn [ : <javaelements> ] (
);
```

The java elements can be :

- Declaration of a simpleorm SrecordInstance class to create :

```
simpleorm “my.class.Name” as <aliasName>
```

- Declaration of a transport class to map to :

```
transport "my.class.Name" as <aliasName>
```

Element declarations are separated by commas.

6.8 Column declaration

Columns are declarated similarly to SQL : the name, followed by the type. There are less types directly supported than in SQL (what SQL, anyway ?). Special constructs allow to force arbitrary SQL text, and you can map to a user-defined SField so all cases can be covered.

6.8.1 Primary key(s)

One or more primary keys must be declared in the SQL part, and also in the java part.

For the sql part, you must use the special statement PRIMARY KEY (…) to declare one or more primary keys. In the java part, you must add the #SPRIMARY_KEY directive (in upppercase). You can add arbitrary literal SQL if your SQL declares primary keys differently (although I didn't see any need to for now). In the future, the PRIMARY KEY should be supported in the column declaration, but for the moment, it isn't.

6.8.2 declaring references

To declare simpleorm references, there is a special construct that uses the "ref" keyword. The syntax to declare it is :

```
<alias>.<ref_field_name> ref <name_of_referenced_record>
```

For example, in the demo, there is a FRIENDS_ID column that references a FRIENDS record. It is declared like this :

```
FRIENDS_ID integer : s, s.ref_FRIENDS ref FRIENDS;
```

In the generated **`org/hmn/jrsd/demo/orm/CDS_gen.java`** file, two java static SField fields will be generated :

```
/**  If LENDED_F = '1', ID of the friend to whom the CD has been lended.  */
static final public SFieldInteger FRIENDS_ID =
  new SFieldInteger(meta, "FRIENDS_ID"   )

  ;
/** Reference field using foreign key 'ref_FRIENDS' to record 'FRIENDS' */
static final public SFieldReference<FRIENDS> ref_FRIENDS =
  new SFieldReference<FRIENDS>(meta, FRIENDS.meta, "ref_FRIENDS", FRIENDS_ID)
  ;
```

6.9 "transport" objects

Some classes are just plain old java objects (aka javabeans) whose only purpose is to convey the data from one part of a machine/application to another place (another machine, another database, etc.). Such objects are often called "value objects" or "value transfer objects" because they just hold values, they don't have behaviour. I

prefer to call them “transport objects”, because transporting data is the very essence of their existence. While simpleorm has elegant mechanisms to avoid such objects, not all applications support the simpleorm API, so there still is a need to use transport objects (this was especially true with previous versions of simpleorm). In jrsd, if you declare a transport object, you can declare the fields that have to be filled with the record's data.

When mapping to a simpleorm record, you can declare additional elements to map to, as a list enclosed in parentheses. Here in the following example, a correspondence will be generated between the field login of s and the field login of t :

```
  transport "org.hmn.gesmat.login.LoginForm" as t,
  simpleorm "org.hmn.gesmat.orm.GM_USER"
(
  login varchar(32) : s(t);
```

If the name used in t is different, just add it explicitly in the declaration using the dot notation :

```
  login varchar(32) : s(t.myLogin);
```

You can use more than one transport object, here is a more elaborate example :

```
  transport "org.hmn.gesmat.login.LoginForm" as t,
  transport "org.hmn.gesmat.GmUser" as u,
  simpleorm "org.hmn.gesmat.orm.GM_USER"
    extends abstract "org.hmn.gesmat.orm.GM_USER_gen"
      as s
(
  /* unique key, generated by the gm_user_id_seq sequence */
  gm_user_id integer : s #usesequencegen;
  /* user */
  //login is mapped to 2 different transport objects
  login varchar(32) : s(t.login, u.userLogin);
```

<<TODO describe readFrom and writeTo>>

6.10 Comments

Comments that are enclosed between “/*” and “*/” are recorded, and used in the generated files (both java and sql). You can also generate the code for Oracle that will add the corresponding column comments. You can only have one comment of this sort per table/column definition.

The comments that start with “//” are simple comments and are completely ignored.

7 The generated files

7.1 SQL files

Every jrsd table declaration has a version number, and a checksum number that I call the “sql checksum” : every name and every type (varchar, integer, etc.) participate in the calculation of this checksup (something similar to the *serialVersionUID*

used in java serialization). The name of a generated SQL file for a table includes the version number AND the digest in its file name. If a file exists that has the same prefix (name + version) but a different digest (let's call it file “A”), the age of file “A” is checked. If file A's age is less than 24 hours, it is deleted, and the new file is written. If file A is older than 24 hours, and , the target file has a different digest, this is considered an error and an exception is thrown. This mechanism is there to protect the database from unattended model changes. Therefore, if you add or remove a column in your jrsd descriptor later, you must either increment the table's version number, or delete manually the previous SQL file. This naming scheme is cumbersome, but has saved me from numerous problems when maintaining database code.

7.2 java files

For each “simpleorm <class_name>” statement, a SrecordInstance class is generated. For each column that has been mapped to the SrecordInstance, a corresponding field is generated, with the corresponding getters and setters. This makes the work with IDEs (Eclipse, Netbeans, etc.) much easier. You can add literal code to the generated file, or you can extend it to add further methods. Adding literal code is only necessary to change things in the static fields. For everything else, extending is fine. By convention, I name generated file by appending a “_gen” suffix; the class that extends them has the name of the table. See the CDS_gen and CD classes in the demo for an example.

7.3 Xml files

Sarting with v.2.8 there is the possibility of producing an xml file for further automated processing. There is an example “to-liquibase.xsl” stylesheet to demonstrate this transformation from jrsd xml to the liquibase xml format. The export is rather crude, and for the moment some attributes will certainly need to be checked/updated. To enable xml output use the command line switch -emitxml, or the ant jrsd task attribute emitxml=”true”. The base dir for the generated xml files defaults to the current user directory; it can be changed with the -xmlroot <path> command line argument, or the xmldir=”<path>” task attribute.

8 Installation

The distribution is composed of 2 archives : jrsd-2.8.jar and jrsd-2.8-rt.jar

The “-rt” jar is the runtime path and will be used at runtime. It the only jar that has to be present in the final project. The size of the runtime is approximatively 80 K, but it uses log4j, which is around 350 K.

The other jar is the precompiler, and is the one that generates the source files from the specification file.

These archive have 2 dependencies : antlr-2_7_6.jar and log4j-1.2.8.jar

9 Usage from the command line

jrsd -srcroot src -sqlroot sql mapping-file.jrsd [mapping-file2.jrsd ...]

10 The .jrsd types

The booleanchar type

If you declare a field of booleanchar type, it will be a field represented as a CHAR column of width 1, with the first char meaning true, the other one meaning false.

Example of declaration :

```
ALLOWED BOOLEANCHAR("1", "0") : s;
```

Here the generated sql will be :

```
ALLOWED char(1),
```

and the generated java will be :

```
  static final public SFieldBooleanChar ALLOWED =
    new SFieldBooleanChar(meta, "ALLOWED", "1", "0" );

  public Boolean getALLOWED() {
    return (Boolean)getObject(ALLOWED);
  }

  public void setALLOWED(Boolean v) {
    setObject(ALLOWED, v);
  }
```

11 More advanced .jrsd

11.1 ANTLR grammar

There are lots of options and possibilities, not all are documented yet. For the very detailed syntax, the ANTLR grammar, in **`g/jrsd.g`** is the reference. ANTLR grammars are not too hard to read.

11.2 Quoting of names

The default is to enclose column and table names in quotes only when “necessary”. The rule is : no quoting is necessary if the name starts with a letter and is followed by nothing, letters, digits or underscores. If the special directive **#quote** is used in the SQL part of a table definition, quoting is forced for names. If the special directive **#unquote** is used in the SQL part of a table definition, quoting is disabled for all the names.

11.3 Parameterized types : boolean(...)

booleanchar("1", "0")

11.4 Extends : Using the generated class as a superclass

<TODO>

11.5 Extends abstract

<TODO>

11.6 Explicit simpleorm field type declaration

<TODO>

11.7 Double colon (“::”) usage

Used when the constructed object is assignable to a variable of another class, but is not the same. Example :

```
DESCR varchar(32) : s TruncatedString::String;
```

Will generate :

```
static final public SFieldString DESCR =
    new SFieldTruncatedString(meta, "DESCR"     , 32  )
```

The `simpleorm.dataset.SFieldTruncatedString` class is part of jrsd2 and extends `simpleorm.dataset.SFieldString`.

11.8 field prefixes

When you declare a column, everything gets the same name by default : the column's name, the simpleorm field, and this name is used for the setter and the getter. Some will like this approach (I do), others won't. And in some rare cases, this may cause trouble. For example when you have a field “name”, jrsd will generate the field “SfieldString name”, and the getter “String getName()”. If you look for the “name” using reflexion, you may find a property named “name”, and a getter “getName()” for the property “name”. To avoid ambiguities like this and as a matter of taste, there is a directive introduced in jrsd 2.6 that specifies a prefix to add to all fields. This directive, “field prefix”, must appear at the start of the spec. For example if you use the directive :

```
field prefix “fld”;
```

Every generated field declared in the spec will start with the prefix “fld”.

11.9 declaring a reference

Use the keyword « ref », followed by the table that is referenced. There is a limitation when a reference is declared in this way : the name of the column is also used for the

field name in the produced code.

Example :

```
table MASTER_TABLE version 1 :
  simpleorm "org.hmn.jrsd.test.MASTER_TABLE" as s
(
  MASTER_TABLE_ID integer : s;
  NAME varchar(30) : s;
  PRIMARY KEY(MASTER_TABLE_ID);
);

table DETAIL_TABLE version 1 :
  simpleorm "org.hmn.jrsd.test.DETAIL_TABLE" as s
(
  DETAIL_TABLE_ID integer : s;
  DESCR varchar(30) : s;
  MASTER_TABLE_ID integer : s, s.ref_MASTER_TABLE ref MASTER_TABLE;
  PRIMARY KEY(DETAIL_TABLE_ID);
);
```

When declaring reference fields, you must also declare the field itself, or Simpleorm will complain at runtime that it does not find the referenced foreign key.

Wrong :

```
  MED_SERV_OU_ID VARCHAR(32) : s.ref_MED_SERV_OU ref MED_SERV_OU;
```

Right :

```
  MED_SERV_OU_ID VARCHAR(32) : s, s.ref_MED_SERV_OU ref MED_SERV_OU;
```

Two fields will be generated :

```
  /** ID of the service O.U. this O.U. is attached to */
  static final public SFieldString MED_SERV_OU_ID =
    new SFieldString(meta, "MED_SERV_OU_ID"     , 32  )

    ;
  /** ID of the service O.U. this O.U. is attached to */
  //
  static final public SFieldReference<MED_SERV_OU> ref_MED_SERV_OU =
    new SFieldReference<MED_SERV_OU>(meta, MED_SERV_OU.meta, "ref_MED_SERV_OU",
MED_SERV_OU_ID)
    ;
```

Note here that the field name is not really the referenced field, but the name of the field itself. This is on purpose because I wanted to be able to use the field several times to address several tables. If you don't want this behavior, declare the field manually

Tweaking the generated types

When you declare a field for Simpleorm, the field type is inferred from the declared type in the SQL part. This can be insufficient, in very rare cases. One example is the current use of Blob(s) in Simpleorm, which still needs some improvement. The syntax of jrsd allows you to declare an extra type to use for the simpleorm field, and even a different type that will be used for the data that will be passed to and from the accessors.

For an example, see the part that decribes Blob usage.

11.10 Adding literal code for sql

The column is declared with a name and a type, and this serves to generate the SQL DDL statement to declare that column. However in some cases this is not satisfactory, because the SQL must be very specific. For this special case there is a syntax to declare literally the SQL code that will be used. The sql code is declared inside curly brackets, and is copied as it is in the SQL file (no other text is generated for that column). Here is an example for MySql :

```
MOD_DT TIMESTAMP {MOD_DT DATETIME} : s ;
```

This declares a column MOD_DT of type TIMESTAMP. For this column, a corresponding `SfieldTimestamp` will be generated in the java code, however in SQL, if there was no literal code, a column of type TIMESTAMP would be generated, and unfortunately a column of type TIMESTAMP sets its value to the current date and time if you insert a null value into it, and this is not what we expect (most of the time). The type that corresponds better is DATETIME, thus the declaration inside curly brackets : **`{MOD_DT DATETIME}`** which gives the correct statement for the SQL generation.

Similarly, there are other places where you can add literal SQL : at the start of the table's declaration (after the version number), before and after the end of the table's declaration (the closing parenthesis), you can add literal code that will be placed at the end of the table's SQL, before or after the closing parenthesis. This gives you a great control, ant is useful to add special clauses to the table's declaration, or to add extra index clauses, for example. To see all the possibilities, consult the ANTLR grammar.

11.11 Adding literal code for java

Using the same mechanism than for literal sql, you can add java code that will get copied literally to the generated java code. The code is inserted after the “meta” field declaration, this allows the referencing of the fields that are declared before the “meta” field. Inside the block, only two escape chars are possible : a “}” must be replaced by a “\}”, and a “\” must be replaced by a “\\”. See the demo's PAT_VALUE descriptor for an example.

11.12 Column types and simpleorm field types

Here is a correspondence between the column type that you declare (case-insensitive), and the Simpleorm field type that is generated :

PRIMITIVE_BOOLEANBIT	BooleanBit
PRIMITIVE_BOOLEANCHAR	BooleanChar
INT	Integer
PRIMITIVE_LONG	Long

PRIMITIVE_DOUBLE	Double
BOOLEANBIT	BooleanBit
BOOLEANCHAR	BooleanChar
INTEGER	Integer
LONG	Long
DOUBLE	Double
VARCHAR	String
CHAR	String
NUMBER	Long
NUMBER(n)	BigDecimal
BIGDECIMAL	BigDecimal
DATE	Date
TIMESTAMP	Timestamp
TIME	Time
BYTES	Bytes
BLOB	Blob
CLOB	Clob
OTHER(XXX)	XXX

The difference between LONG and PRIMITIVE_LONG is that for LONG, accessors for “Long” (the Object type) will be generated, whereas for PRIMITIVE_LONG, accessors for “long” (the primitive type, notice the lowercase “L”) will be generated. This is the same for other PRIMITIVE_xxx types.

11.13 Using directives

Directives are case-sensitive. simpleorm directives are uppercase, non-simpleorm directives are lowercase

simpleorm directives :
(See simpleorm doc for more information)

```
#SPRIMARY_KEY
#SMANDATORY
```

```
#SNOT_OPTIMISTIC_LOCKED
#SDESCRIPTIVE,
#SUNQUERIED
```

“Non-simpleorm” directives (they don't correspond to a simpleorm constant, but generate code for simpleorm, or control code generation otherwise)

`#usesequencegen`
Declare in simpleorm the use of a sequence to generate new ids. The name of the sequence is fixed; it is the name of the field plus « _GEN ». Example of generated java code for the field that corresponds to the column « FRIEND_ID » :
`.setGeneratorMode(SGeneratorMode.SSEQUENCE, "FRIEND_ID_SEQ")`

`#useinsertgen`
Declare in simpleorm the use of the « `insert` » generator mode (used for MySql mainly, and some others)

`#useselectmaxgen`
Declare in simpleorm the use of the « `selectMax` » generator mode, that uses `select max(colname) + 1` to get new ids.

`#noaccessors`
Don't generate getters & setters for this column.

11.14 Declaring key generators

When you declare a primary key column, the column's value can be automatically generated by the database or by simpleorm (see simpleorm doc for more information).

The simplest usage is to declare an integer field, for which automatic key generation is now supported by most databases, and to use the most appropriate directive (#usesequencegen for Oracle, #useinsertgen for MySql, #useselectmaxgen for others, mostly). Note that when using #usesequencegen, the name of the sequence is fixed, and is the name of the field followed by _GEN. This is sufficient in most cases; however if a specific name is really needed, the name of the sequence can be changed using the simpleorm API inside a static initialization block, declared inside the literal java code part. Ex :

```
simpleorm "test.MYTABLE" as s
{
  static {
    MYFIELD.setGeneratorMode(SGeneratorMode.SSEQUENCE, "MY_SEQ");
  \}
}
```

This works because the literal code is inserted after the field's declarations.

11.15 Quoting

The directive “quote” and “unquote” in a column's def. Default is to quote if necessary. No quoting is necessary if name start with a letter and contains only letters,

digits, and the underscore character.

11.16 Using a custom type

Although seldom used, simpleorm enables you to create your own Sfield class. Jrsd supports this. To use this feature the syntax “other(“MyCustomType”)” is used. The generated source uses the type SfieldMyCustomType. Here is an example that uses a “BigInteger” type :

First we declare the SfieldBigInteger class, with a constructor that cant take only two values (this is what jrsd will generate).

```
package simpleorm.dataset;

import java.math.BigDecimal;
import java.math.BigInteger;
import java.sql.ResultSet;

import simpleorm.utils.SException;

/**
 * Special implementation of SFieldBigInteger, to avoid rounding
errors
 * when dealing with ens indexes.
 * @author hk
 * (currently under development)
 */
public class SFieldBigInteger
extends SFieldScalar
{
  static final long serialVersionUID = 1L;

  private int precision = 0;
  private int scale = 0;

  public SFieldBigInteger(SRecordMeta meta, String columnName,
SFieldFlags... pvals)
  {
    this(meta, columnName, 38, 0, pvals);
  }

  /**
   * Note that precission and scale parameters only affect how the
tables are
   * created. The scale that is actually returned is up to JDBC. And
then you
   * are responsible for dealing with rounding issues.
   */
  public SFieldBigInteger(SRecordMeta meta, String columnName,
          int precission, int scale, SFieldFlags... pvals) {
      super(meta, columnName, pvals);
      this.precision = precission;
      this.scale = scale;
  }
```

```
  public int getPrecision() {
      return precision;
  }
  public int getScale() {
      return scale;
  }

  public Object queryFieldValue(ResultSet rs, int sqlIndex) throws
Exception {
      /**
       * CHANGED to add second parameter
       */
      BigDecimal bd = rs.getBigDecimal(sqlIndex);
      if (rs.wasNull()) // ie. last operation!
          return null;
      else {
        BigInteger res = bd.toBigInteger();
        return res;
      }
  }

  protected Object convertToDataSetFieldType(Object raw)
  throws Exception
  {
      if (raw instanceof BigDecimal)
          return ((BigDecimal) raw).toBigInteger();
      if (raw == null)
          return null;
      if (raw instanceof Number)
          return new BigInteger(raw.toString());
      // ## This will loose precission for longs, but Java does not
      // provide an easy way to convert longs to BigDecimals! There
is
      // certainly no Number.bigDecimalValue.
      if (raw instanceof String) {
          BigInteger val = new BigInteger(raw.toString());
          return val;
      }
      throw new SException.Data("Cannot convert " + raw + " to
BigInteger.");
  }

  /**
   * This is basically SQL 2, and fairly database
   * independent, we hope. Note that "BIGDECIMAL" for Oracle means a
text
   * field that can contain over 2K characters!
   */
  @Override
  public String defaultSqlDataType() {
      return "NUMERIC("
              + getPrecision()
              + "," + getScale()
              + ")";
  }

  public boolean isFKeyCompatible(SFieldScalar field) {
      if (!(field instanceof SFieldBigInteger))
```

```
            return false;
        SFieldBigInteger biField = (SFieldBigInteger) field;
        if (biField.getPrecision() != this.getPrecision()) return
false;
        if (biField.getScale() != this.getScale()) return false;
        return true;
    }

    @Override
    public
    int javaSqlType() { return java.sql.Types.NUMERIC; }

}
```

We can then declare the column of type « other », and use the sql we want for the column :

```
  IXLO other("BigInteger") { IXLO NUMBER(38, 0) } : s;
```

The type BigInteger is not known by default, we must add an include statement right after the jrsd preamble :

```
jrsd 2;

import "java.math.BigInteger";
```

Default names

When you omit some names, defaults are used. For example, writing
`foo_bar  integer : s(t);`

Is the same as writing
`foo_bar integer : s.foo_bar(t.foo_bar);`

This makes the code more readable and it is advised to keep the same names in all objects to avoid bugs that are difficult to find.

11.17 Field lists

You can declare a field list, and declare in the jrsd file your fields they will be added to this list. This can be useful to adress only a subset of some values.

11.18 Using LOBs

11.18.1 Why it ~~is~~ was difficult

Database fields usually have a fixed length, and are kept small. This is because their values are stored in indexes, and in structures that are frequently re-allocated, when the database gets reorganized. However there is a need to store bigger « values », like

binary data that represents a picture or a video, or a long text (like an entire xml document). To store these big values, databases like Oracle, MySql, and others use reserved areas, that are manipulated outside of the internal record structure. In the record structure, there is a pointer to this area. A large area that contains characters is called a CLOB (for Character Large OBject), and a large area that contains binary data is called a BLOB (for Binary Large Object). These elements are not like regular scalar values (numbers, strings, dates, …). Access to theses elements via JDBC is tricky, because the API looks just like the API that is used to retrieve Strings, Integers, etc. However, if you try to manipulate LOBs like usual scalar values, you start to get errors and incomprehensible behavior.

Things get even more confusing when you look at the various docs available. If you consult the Java API documentation, (cf. docs/guide/jdbc/blob.html), you don't get every aspect of LOBs, especially not for Oracle. There is no code, for example, that creates a new LOB. If you look at Oracle samples as supplied by Oracle for version 9, (cf. advanced_jdbc_samples/LOBSampleReadme.html), you see that the creation of a BLOB or CLOB occurs in SQL, not in the JDBC API, via the instruction EMPTY_BLOB() (resp. EMPTY_CLOB()).So the part that creates the LOB is database-dependent; which is not very much in the spirit of JDBC. Actually, you *can* create a new BLOB indirectly, with JDBC, look at

`org.hmn.jrsd.test.BlobTests2.testRegularBlobCreation()`

for an example.

When you look at the various samples for Oracle, you see that reading / writing to a LOB requires :

1) that you select and lock the row that contains the handle to the LOB, using the FOR UPDATE clause
2) that you use `blob.getBinaryStream()`, resp. `blob.setBinaryStream()`, to get a stream to read or to write data to/from the BLOB

This is rather complex (but it works). Jdbc3 supports writing bytes directly via setBytes(), but this requires a (relatively) recent driver. And also if the Lobs you manipulate are rather large, this can consume a lot of memory. And unfortunately, this doesn't work directly in Simpleorm, because when you try to write bytes using field.setBytes(), Simpleorm will call PreparedStatement.setObject(), not PreparedStatement.setBytes(), and the database may complain that it needs a Blob object, not a byte array (recent oracle 10 drivers handle the array correctly by converting it automatically; see the test `org.hmn.jrsd.test.BlobTests3.testPsSetBytes()`). So the “official” way is to retrieve a Lob object and to manipulate it. To help you do this, jrsd2 has the methods org.hmn.jrsd.rt.SormUtils.readFromBlob(Blob), and org.hmn.jrsd.rt.SormUtils.writeToBlob(byte[], Blob). To select the Blob, you can use the method

```
org.hmn.jrsd.rt.SormUtils.selectExtra(SSessionJdbc, SRecordInstance,
String, ResultSetColVisitable)
```

and the object

`org.hmn.jrsd.rt.BlobCollectVisitor`

to get access to the Blob. Ex :

```
BlobCollectVisitor visitor = new BlobCollectVisitor();
SormUtils.selectExtra(s, rec, "SOMEBLOB", visitor);
Blob clb = visitor.getBlob();
```

See for an example :

`org.hmn.jrsd.test.BlobTests3.testUpdateSelectExtra()`
If you can use a recent jdbc driver (and most of the time, you can), PreparedStatement.setBytes() and ResultSet.getBytes() will work, so using `SFieldBlob` will work for the reading part, and make the code much simpler (in Oracle this only works reliably if JDBC driver for Oracle 10, is used, as there were bugs in the handling of `ps.setBytes()` in previous versions). For the writing part, we will have to wait until Simpleorm calls setBytes instead of setObject.

11.18.2 How to do it (the simple way)

- The key is to use the most recent JDBC driver available. Be sure the driver supports JDBC3. This was a problem a few years ago, but now it isn't.
- Declare the field as *SUNQUERIED*, ex :

```
static SFieldBlob SOMEBLOB =
  new SFieldBlob(meta, "SOMEBLOB", 500000, SfieldFlags.SUNQUERIED);
```

- Use the select mode *SALL* to select the record, this will read all fields even the ones that are normally unqueried :

```
rec = s.findOrCreate(TEST_BLOB3.meta, SSelectMode.SALL, 7);
```

- Read the bytes in the field, just like a scalar value
- If you use Oracle 10, the commit will write back the bytes correctly
- To write the bytes differently (if update doesn't work), don't update the record, but use the utility method SormUtils.writeToBlob(), after locking the row. Ex :

```
MYTABLE rec = s.find(MYTABLE.meta, SQueryMode.SFOR_UPDATE, 2);
SormUtils.writeToBlob(s, rec, "SOMEBLOB", theBytes);
s.flushAndPurge(rec);
```

- If your database is not Oracle, test the creation and the update, using small LOBs AND large LOBs (like 1K and 1M), because some bugs may only appear when the data is large, and this area seems less tested than others in simpleorm.
- In 99% of the cases, using byte arrays is sufficient, but some testing should be done.
- For more complex needs, see the paragraph above to select the Clob or the Blob.

11.18.3 The other problem is CLOBs

Clobs hold characters, not bytes, just like String. 99% of the time, you will want to

store a string directly, there is no necessity to copy the data from a stream. Some JDBC drivers (Oracle for example), support writing to the CLOB using PreparedStatement.setString() (actually, even PreparedStatement.setObject(String[]) seems to work in Oracle, but I don't have seen this documented). So to use a CLOB, declare it as SfieldClob2 (this class was added in jrsd2, it is not part of Simpleorm), reading from the CLOB field will get the CLOB's handle, and read from it. However, Simpleorm writes PreparedStatement values using setObject(), this will work with the Oracle 10 driver, but may not work for others as the database sometimes expects exactly a Clob object. There is a utility method that updates a column using setString() explicitly, you can use that method to update the CLOB. This is particularly useful to initialize the CLOB, it is much simpler than finding the specific clause that creates an empty CLOB (check your database for details). You must acquire a lock on the record before you modify a CLOB, so you should first retrieve the row using the SFOR_UPDATE clause. Also most of the time you will not want to read those big CLOB values everytime you select the rows, so the field will be declared as SUNQUERIED, so to select it, SALL must also be used. Example (simplified) from `org.hmn.jrsd.test.ClobTests2.testSetAsString()` :

```
String randomContent = makeRandomString();
SSessionJdbc s = SSessionJdbc.open(ds, "testSetAsString");
s.begin();
rec = s.find(TEST_CLOB2.meta, SQueryMode.SFOR_UPDATE, 7);
SormUtils.updateColumnAsString(
    s, rec, "SOMECLOB", randomContent);
s.commit();
//re-read
s.begin();
rec = s.findOrCreate(
   TEST_CLOB2.meta, SSelectMode.SALL, SQueryMode.SFOR_UPDATE, 7);
readContents = rec.getString(TEST_CLOB2.SOMECLOB);
s.commit();
```

11.18.4 How to declare in jrsd2

- The current SfieldBlob implementation (rev 845 of Simpleorm) does not return a Blob actually, but rather the bytes directly. For this reason, jrsd2 would produce incorrect accessors, so this field must be declared differently.
- Declare as a BLOB field, but declare the returned type as bytes (which will get replaced by “byte[]”); also use a directive to set the Simpleorm SUNQUERIED flag :

```
SOMEBLOB BLOB : s Blob::bytes #SUNQUERIED;
```

- If you don't use an Oracle 10 driver, don't use use the setSOMEBLOB() generated accessor, as it takes a byte[] argument, but jdbc will usually expect a Blob, so this will not work. Use SormUtils.writeToBlob() instead, as explained above. Ex :

```
MYTABLE rec = s.find(MYTABLE.meta, SQueryMode.SFOR_UPDATE, 2);
SormUtils.writeToBlob(s, rec, "SOMEBLOB", theBytes);
s.flushAndPurge(rec);
```

- For CLOBs the situation is different, as Simpleorm does not currently propose a SfieldClob. You can use SfieldClob2, supplied with jrsd2, but if you should declare the accessors yourself, because jrsd doesn't know how to generate them correctly, and if you don't use an Oracle 10 driver, declare only a getter. To set the value of the CLOB, see the chapter about CLOBs. Ex of declaration :

```
  SOME_LONG other("Clob2")
              {SOME_LONG CLOB} : s #noaccessors #SUNQUERIED;
...
  public String getSOME_LONG() { return getString(SOME_LONG); }

  public void setSOME_LONG(String str) { setString(SOME_LONG, str); }
```

11.19 Utilities in jrsd2 package

11.19.1 simpleorm.utils.SLogLog4jBridge

11.19.1.1 Introduction

(introduced in jrsd 2.5) This utility serves to control the logging that is output to a log4j Logger.

Every time simpleorm needs a logger, it instanciates one using Slog.newSLog() which uses the slogClass to make that instantiation. SlogLog4jBridge, when you call useSLogLog4Bridge() puts its class in the slogClass field, so that instances of SlogLog4jBridge are used as loggers.

The configuration of the logger is stored in another object, the Configuration object. It controls, for each Slog logging operation, what log4j level is used to emit to log4j. It has as a default the most usual setting, but everything can be overridden. When a SlogLog4jBridge is created, its configuration is the default configuration. This default configuration is a singleton, and changing it changes the behavior of all objects that use it.

You can create you own Configuration objects, and use them to configure you logger. When is this needed ? Well, for each new connection, a new Slog instance is created using Slog.newSLog(). To change the settings for the new connection only, you can tell that the new connection's logger has to use the given configuration, by calling the configuration's method applyTo(). Example :

```
Configuration myConf = new Configuration();
// set the wanted levels in myConf
...
SsessionJdbc s = SsessionJdbc.open(ds, "myConn");
myConf.applyTo(s);
```

The same conf object can be used for several configurations.

11.19.1.2 Simple configuration

To use the SlogLog4jBridge you just have to execute the static method useSLogLog4jBridge() as soon as possible (in the main() method, for example). After that, you can change a few levels in the default configuration. These levels will be used in all places that use the logger. Ex :

```
SlogLog4jBridge.useSLogLog4jBridge();
```

You can change levels at the default configuration, this will change the level everywhere. Ex :

```
SLogLog4jBridge.getDefaultConfiguration()
    .setLevel(SLogLog4jBridge.LN_CONNECTIONS, "info");
```

11.19.1.3 More elaborate configuration

After having used the simple configuration above, we will show how to configure loggers for each session. Here in the following example MyClass has a factory method, that produces connections, whose logger has a few custom levels.

```
public MyClass {
  DataSource ds;
  Configuration mySpecialConf = new Configuration();

  public MyClass(DataSource ds) {
    this.ds = ds;
    //the special conf is that queries are emitted with the "info"
    //level
    mySpecialConf.setLevel("queries", "info");
  }

  /**
   * Make a connection, the connection will use a logger that
   * has our special configuration
   */
  public SsessionJdbc makeConnection(DataSource ds, String name)
  {
    SsessionJdbc s = SsessionJdbc.open(ds, name);
    mySpecialConf.applyTo(s);
    return s;
  }
}
```

11.19.1.4 The log4j logger used

The default logger used in a new Configuration instance is the logger attached to `SlogLog4jBridge`. The "lg" field is public, so you can change this very easily. From the SlogLog4jBridge instance you can also use setLogger() to change the logger in the underlying configuration instance.

11.19.1.5 Deriving Configuration objects

The configuration instances are shared between object that thy apply to. If you change one level, this change will apply to all objects that use that conf. If you just want to

change some levels for a few special connections, you can clone an existing Configuration object and change the levels you want, Configuration implements Cloneable.

11.20 DbOps

<TODO>

11.21 SQL expression objects

This is somewhat redundant with simpleorm; There is a `org.hmn.jrsd.rt.WhereHelper` class that help to build a correct "where" clause, when some parameters can be null.The expression must be entered as a combination of SqlExpr derived objects (AndExpr, OrExpr, BinOpExpr, ListExpr, LiteralExpr).

See `org.hmn.jrsd.test.SqlExprTests.testAndExpr()` for an example.

11.22 FiniteString class

A utility class to control the length of the string from end to end, truncating it if necessary. This is to avoid errors due to excessive length of strings in unimportant columns. This class should not be used for more important columns (strings that serve as ids, for example) where silent truncation is not acceptable.

11.23 JdbcResults

org.hmn.jrsd.rt.rows.JdbcResults has nothing to do with simpleorm. In fact I used it well before simpleorm. It is useful to get the result of a query in a "compact" form for serialization. This is very simple, but is somehow redundant with simpleorm. The rows, columns, cells are accessible as simple collections and with javabean semantics, so with libraries that requires this (like stringtemplate) this is more convenient than simpleorm.

11.24 Simpleorm extra utils, Jdbc utils

Jrsd proposes several small methods, rather trivial, that I use frequently, so they don't need to be recoded every time. These methods are in the SormUtils and JrsdRtUtils class.

11.25 Integration with ant

You can define a task in ant, that will do all the precompilation.

First, declare a classpath that references the 4 jars that are necessary :

```
<path id="jrsd_cp">
  <pathelement location="${log4j.jar.path}" />
  <pathelement location="${jrsd.jar.path}" />
  <pathelement location="${antlr.jar.path}" />
  <pathelement location="${st.jar.path}" />
</path>
```

Then, use that classpath to declare a jrsd task :

```
<!-- declare the jrsd task -->
<taskdef name="jrsd" classname="org.hmn.jrsd.JrsdTask"
         classpathref="jrsd_cp" />
```

You can now use the jrsd task to precompile all your jrsd to sql files and java classes :

```
<jrsd srcdir="${src.dir}" sqldir="${sql.dir}">
  <fileset dir="${jrsd.dir}">
    <include name="**/*.jrsd"/>
  </fileset>
</jrsd>
```

With the distribution comes an example of a typical ant build file. See “References” for more detailed information.

11.26 References

11.26.1 Command line switches

-srcroot <path_to_dir> : root location of generated java source files, Mandatory.

-sqlroot <path_to_dir> : root location of generated sql files, Mandatory.

-xmlroot <dir> : base dir for the generated xml files. Optional. Defaults to “.” (the current user directory). Default is to exit with an error if the output file already exists and is older than 1 day.

-log <path_to_log_file>

-fileext : file extention of the generated sql files. Default is “sql”

-overwrite : overwrite SQL files even if they are older than one day (use with caution, see above)

-forcefieldsuc : force field names to uppercase

-forcefieldslc : force field names to lowercase

-genoracomments : generate oracle comment-creation ddl instructions for column and table comments, in Oracle syntax.

-debug : turn on output of debugging messages

-help : show a help screen, print help and exit

-encoding <encodingname> : force the encoding. This is recommended if you include the generated SQL into jars to load them as resource, the recommended encoding is then “UTF-8”. The default is the platform's default encoding, see the Charset class for more information.

-nojava : turn off java code generation

-nosql : turn off sql code generation

-debug : switch on debug mode

-emitxml : enables the emission of xml files built from the table sql specs.

11.26.2 Ant task attributes

- **`sqldir`** : root path of generated SQL files
- **`srcdir`** : root path of generated java source files
- **`xmldir`** : root path of generated xml files. Defaults to “.”
- **`genoracomments`** : generate oracle SQL instructions to declare comments for tables and columns.
- **`overwrite`** : allow overwriting of sql files of same version and different sql checksum, even if they are more than 1 day old.
- debug : enable debugging
- **`fileext`** : file extension of generated sql files (defaults to .sql)
- **`nojava`** : disable generation of java source files
- **`nosql`** : disable generation of sql files
- **`forcefieldsuc`** : force the case of the generated fields to uppercase
- **`forcefieldslc`** : force the case of the generated fields to lowercase

An ant *fileset* is used to specify the jrsd files that must be processed.

Example of ant task :

```
<jrsd srcdir="src" sqldir="src/sql" xmldir="src/xml" emitxml="true">
  <fileset dir="jrsd-src"/>
</jrsd>
```

x

11.26.3 Supported types for SQL

char(n), varchar(n), bigdecimal, integer, int, primitive_long, long, primitive_double, double, date, timestamp, time, blob, clob, blob(n), clob(n), primitive_booleanbit, booleanbit, bytes(n), number, other(str), primitive_booleanchar(t,f), booleanchar(t,f)

11.26.4 Supported types for simpleorm

int, integer, long, primitive_long, double, primitive_double, boolean, primitive_boolean, date, bigdecimal, ref <field>

Table of contents

12 History

2011-09-02 hk now documenting v.2.8. Xml emission. Updated command line and ant descriptions
2010-12-27 hk more examples and explanations. Explanations on generators and literal code
2010-08-25 hk added field prefix directive
2010-08-02 hk more explanations on the ant task
2010-08-02 hk now documenting v. 2.5
2010-02-22 hk explanations about LOBs
2009-06-11 hk clarifications about references