Reverse Engineering Plugin - Reference Documentation

Authors: Burt Beckwith

Version: 0.1

Table of Contents

1 Introduction to the Reverse Engineering Plugin

The Reverse Engineering plugin reads database table information using JDBC and uses the schema data to create domain classes. This is a complex problem and the plugin is unlikely to get things 100% correct. But it should save you a lot of work and hopefully not require too much tweaking after the domain classes are generated.
The plugin uses the Hibernate Tools library, with custom code to generate GORM domain classes instead of Java POJOs.

1.1 Getting Started

The first step is installing the plugin:

```
grails install-plugin reverse-engineer
```

There are no required initialization steps but there are some configuration options described in the next section.

2 Configuration

There are several configuration options for the plugin.

Core properties
These configuration options are the one you're most likely need to set. They include the package name of the generated files (`grails.plugin.reveng.packageName`), and information about which side of a many-to-many is the 'belongsTo' side (`grails.plugin.reveng.manyToManyBelongsTos`).

Property	Default	Meaning
grails.plugin.reveng.packageName	application name	package name for the generated domain classes
grails.plugin.reveng.manyToManyBelongsTos	none	a Map of join table name -> belongsTo table name to specify which of the two domain classes is the 'belongsTo' side of the relationship

Inclusion/exclusion properties
These configuration options let you define which tables and columns to include in processing. By default all tables and columns are processed.
If you specify tables to include (`grails.plugin.reveng.includeTables`, `grails.plugin.reveng.includeTableRegexes`, and/or `grails.plugin.reveng.includeTableAntPatterns`) then only those will be used. If you specify any of these options then all of the table exclusion options are ignored; this is useful when you have already run the reverse engineer script but want to re-run it for only a subset of tables.
If you specify tables to exclude (`grails.plugin.reveng.excludeTables`, `grails.plugin.reveng.excludeTableRegexes`, and/or `grails.plugin.reveng.excludeTableAntPatterns`) then any matching tables will be ignored.
You can also specify columns to exclude (`grails.plugin.reveng.excludeColumns`, `grails.plugin.reveng.excludeColumnRegexes`, and/or `grails.plugin.reveng.excludeColumnAntPatterns`) and any matching columns will be ignored.
These options can be used with table include rules or table exclude rules - any tables that are included or not excluded will have their columns included or excludede based on these rules.
One addition property, `grails.plugin.reveng.manyToManyTables`, doesn't affect whether a table is processed, but rather determines whether a table that won't look like a many-to-many table to the Hibernate Tools library (because it has more than two columns) is considered a many-to-many table.

Property	Default	Meaning
grails.plugin.reveng.includeTables	none	a List of table names to include for processing
grails.plugin.reveng.includeTableRegexes	none	a List of table name regex patterns to include for processing
grails.plugin.reveng.includeTableAntPatterns	none	a List of table name Ant-style patterns to include for processing
grails.plugin.reveng.excludeTables	none	a List of table names to exclude from processing
grails.plugin.reveng.excludeTableRegexes	none	a List of table name regex patterns to exclude from processing
grails.plugin.reveng.excludeTableAntPatterns	none	a List of table name Ant-style patterns to exclude from processing
grails.plugin.reveng.excludeColumns	none	a Map of table name -> List of column names to ignore
grails.plugin.reveng.excludeColumnRegexes	none	a Map of table name -> List of column name regex patterns to ignore
grails.plugin.reveng.excludeColumnAntPatterns	none	a Map of table name -> List of column name Ant-style patterns to ignore
grails.plugin.reveng.manyToManyTables	none	a List of table names that should be considered many-to-many join tables; needed for join tables that have more columns than the two foreign keys

Other properties
These remaining configuration options allow you to specify the folder where the domain classes are generated (`grails.plugin.reveng.destDir`) and non-standard 'version' column names (`grails.plugin.reveng.versionColumns`).

Property	Default	Meaning
grails.plugin.reveng.destDir	'grails-app/domain'	destination folder for the generated classes, relative to the project root
grails.plugin.reveng.versionColumns	none	a Map of table name -> version column name, for tables with an optimistic locking column that's not named 'version'

3 General Usage

It's most convenient when creating an application to also create the database at the same time. When creating a "greenfield" application like this you can let Hibernate create your tables for you (e.g. using a `dbCreate` value of `create-drop` or `update`). But often the database already exists, and creating GORM domain classes from them can be a tedious process.
This plugin will save you a lot of time by using the JDBC metadata API to inspect your database schema and create domain classes for you, using some assumptions and augmented by configuration options that you set.
The core of the plugin is the reverse-engineer script. There are too many configuration options to support specifying them when running the script, so it takes no arguments and uses configuration options set in `grails-app/conf/Config.groovy`. These are described in section 2.

Environments
You can choose which database to read from by specifying the environment when running the script (like any Grails script). For example to use the development environment settings, just run

```
grails reverse-engineer
```

To use the production environment settings, run

```
grails prod reverse-engineer
```

And if you want to use a custom 'staging' environment configured in `DataSource.groovy`, run

```
grails -Dgrails.env=staging reverse-engineer
```

Re-running the reverse engineering script
If you have new or changed tables you can re-run the reverse-engineer script to pick up changes and additions. This is not an incremental process though, so existing classes will be overwritten and you will lose any changes you made since the last run. But it's simple to define which tables to include or exclude.
As described in section 2, you can use a combination of the `grails.plugin.reveng.includeTables`, `grails.plugin.reveng.includeTableRegexes`, `grails.plugin.reveng.includeTableAntPatterns`, `grails.plugin.reveng.excludeTables`, `grails.plugin.reveng.excludeTableRegexes`, and `grails.plugin.reveng.excludeTableAntPatterns` properties to define which tables to include or exclude.
By default all tables are included, and the plugin assumes you're more likely to exclude than include. So you can specify one or more table names to explicitly exclude using `grails.plugin.reveng.excludeTables`, one or more regex patterns for exclusion using `grails.plugin.reveng.excludeTableRegexes`, and one or more Ant-style patterns for exclusion using `grails.plugin.reveng.excludeTableAntPatterns`.
For example, using this configuration

```
grails.plugin.reveng.excludeTables = ['clickstream', 'error_log']
grails.plugin.reveng.excludeTableRegexes = ['temp.+']
grails.plugin.reveng.excludeTableAntPatterns = ['audit_*']
```

you would process all tables except `clickstream` and `error_log`, and any tables that start with 'temp' (e.g. `tempPerson`, `tempOrganization`, etc.) and any tables that start with 'audit_' (e.g. 'audit_orders', 'audit_order_items', etc.)
If you only want to include one or a few tables, it's more convenient to specify inclusion rules rather than exclusion rules, so you use `grails.plugin.reveng.includeTables`,

`grails.plugin.reveng.includeTableRegexes`, and `grails.plugin.reveng.includeTableAntPatterns` for that. If any of these properties are set, the table exclusion rules are ignored.
For example, using this configuration

```
grails.plugin.reveng.includeTables = ['person', 'organization']
```

you would process (or re-process) just the `person` and `organization` tables. You can also use The `grails.plugin.reveng.includeTableRegexes` and `grails.plugin.reveng.includeTableAntPatterns` properties to include tables based on patterns.
You can further customize the process by specifying which columns to exclude per-table. For example, this configuration

```
grails.plugin.reveng.excludeColumns = ['some_table': ['col1', 'col2'],
                                       'another_table': ['another_column']]
```

will exclude columns `col1` and `col2` from table `some_table`, and column `another_column` from table `another_table`.
You can also use the `grails.plugin.reveng.excludeColumnRegexes` and `grails.plugin.reveng.excludeColumnAntPatterns` properties to define patterns for columns to exclude.

Destination folder
By default the domain classes are generated under the `grails-app/domain` folder in the package specified. But you can override the destination, for example if you're re-running the process and want to compare the new classes with the previous ones.

The `reverse-engineer` script will overwrite existing classes without warning. This is a limitation of the underlying Hibernate Tools library which is used to create the classes.

Many-to-many tables
Typically many-to-many relationships are implemented using a join table which contains just two columns which are foreign keys referring to the two related tables. It's possible for join tables to have extra columns though, and this will cause problems when trying to infer relationships. By default Hibernate will only consider tables that have two foreign key columns to be join tables. To get the script to correctly use join tables with extra columns, you can specify the table names with the `grails.plugin.reveng.manyToManyTables` property. This is demonstrated in the tutorial.
Another problem with many-to-many relationships is that one of the two GORM classes needs to be the 'owning' side and the other needs to be the 'owned' side, but this cannot be reliably inferred from the database. Both classes need a `hasMany` declaration, but the 'owned' domain class also needs a `belongsTo` declaration. So all of your many-to-many related tables need to have the tables that will create the `belongsTo` classes specified in the `grails.plugin.reveng.manyToManyBelongsTos` property. This is demonstrated in the tutorial.

Optimistic locking columns
Hibernate assumes that columns used for optimistic lock detection are called `version`. If you have customized one or more column names, you can direct the script about what the custom names are with the `grails.plugin.reveng.versionColumns` property. This is demonstrated in the tutorial.

4 Tutorial

In this tutorial we'll create a MySQL database and generate domain classes from it.

1. Create your Grails application.

```
$ grails create-app reveng-test
$ cd reveng-test
```

2. Install the plugin.

```
$ grails install-plugin reverse-engineer
```

3. Configure the development environment for MySQL.
Set these property values in the `development` section of `grails-app/conf/DataSource.groovy`

```
dataSource {
   url = 'jdbc:mysql://localhost/reveng'
   driverClassName = 'com.mysql.jdbc.Driver'
   username = 'reveng'
   password = 'reveng'
   dialect = org.hibernate.dialect.MySQL5InnoDBDialect
}
```

Also add a dependency for the MySQL JDBC driver in `grails-app/conf/BuildConfig.groovy`:

```
dependencies {
   runtime 'mysql:mysql-connector-java:5.1.5'
}
```

and be sure to uncomment the `mavenCentral()` repo:

```
repositories {
   grailsPlugins()
   grailsHome()
   grailsCentral()
   mavenCentral()
}
```

4. Create the database.
As the root user or another user that has rights to create a database and configure grants, run

```
create database reveng;
grant all on reveng.* to reveng@localhost identified by 'reveng';
```

5. Create the database tables.
Run these create and alter statements in the `reveng` database:

```
use reveng;
```

```
create table author (
   id bigint not null auto_increment,
   version bigint not null,
   name varchar(255) not null,
   primary key (id)
) ENGINE=InnoDB;
create table author_books (
   author_id bigint not null,
   book_id bigint not null,
   primary key (author_id, book_id)
) ENGINE=InnoDB;
create table book (
   id bigint not null auto_increment,
   version bigint not null,
   title varchar(255) not null,
   primary key (id)
) ENGINE=InnoDB;
create table compos (
   first_name varchar(255) not null,
   last_name varchar(255) not null,
   version bigint not null,
   other varchar(255) not null,
   primary key (first_name, last_name)
) ENGINE=InnoDB;
create table compound_unique (
   id bigint not null auto_increment,
   version bigint not null,
   prop1 varchar(255) not null,
   prop2 varchar(255) not null,
   prop3 varchar(255) not null,
   prop4 varchar(255) not null,
   prop5 varchar(255) not null,
   primary key (id),
   unique (prop4, prop3, prop2)
) ENGINE=InnoDB;
create table library (
   id bigint not null auto_increment,
   version bigint not null,
   name varchar(255) not null,
   primary key (id)
) ENGINE=InnoDB;
create table other (
   username varchar(255) not null,
   nonstandard_version_name bigint not null,
   primary key (username)
) ENGINE=InnoDB;
create table role (
   id bigint not null auto_increment,
   version bigint not null,
   authority varchar(255) not null unique,
   primary key (id)
) ENGINE=InnoDB;
create table thing (
   thing_id bigint not null auto_increment,
   version bigint not null,
   email varchar(255) not null unique,
   float_value float not null,
   name varchar(123),
   primary key (thing_id)
) ENGINE=InnoDB;
create table user (
   id bigint not null auto_increment,
   version bigint not null,
   account_expired bit not null,
   account_locked bit not null,
   enabled bit not null,
   password varchar(255) not null,
   password_expired bit not null,
   username varchar(255) not null unique,
   primary key (id)
) ENGINE=InnoDB;
create table user_role (
   role_id bigint not null,
   user_id bigint not null,
   date_updated datetime not null,
   primary key (role_id, user_id)
) ENGINE=InnoDB;
create table visit (
   id bigint not null auto_increment,
   library_id bigint not null,
   person varchar(255) not null,
   visit_date datetime not null,
   primary key (id)
) ENGINE=InnoDB;
```

```
alter table author_books add index FK24C812F63FA913A (book_id),
add constraint FK24C812F63FA913A foreign key (book_id) references book (id);
alter table author_books add index FK24C812F6CD85EDFA (author_id),
add constraint FK24C812F6CD85EDFA foreign key (author_id) references author (id);
alter table user_role add index FK143BF46A52388A1A (role_id),
add constraint FK143BF46A52388A1A foreign key (role_id) references role (id);
alter table user_role add index FK143BF46AF7634DFA (user_id),
add constraint FK143BF46AF7634DFA foreign key (user_id) references user (id);
alter table visit add index FK6B04D4B4AEC8BBA (library_id),
add constraint FK6B04D4B4AEC8BBA foreign key (library_id) references library (id);
```

6. Configure the reverse engineering process.
Add these configuration options to `grails-app/conf/Config.groovy`:

```
grails.plugin.reveng.packageName = 'com.revengtest'
grails.plugin.reveng.versionColumns = [other: 'nonstandard_version_name']
grails.plugin.reveng.manyToManyTables = ['user_role']
grails.plugin.reveng.manyToManyBelongsTos = ['user_role': 'role', 'author_books': 'book']
```

7. Run the reverse-engineer script.

```
grails reverse-engineer
```

8. Look at the generated domain classes.

Author and Book domain classes.
The `author` and `book` tables have a many-to-many relationship, which uses the `author_books` join table:

```
create table author (
   id bigint not null auto_increment,
   version bigint not null,
   name varchar(255) not null,
   primary key (id)
) ENGINE=InnoDB;
create table author_books (
   author_id bigint not null,
   book_id bigint not null,
   primary key (author_id, book_id)
) ENGINE=InnoDB;
create table book (
   id bigint not null auto_increment,
   version bigint not null,
   title varchar(255) not null,
   primary key (id)
) ENGINE=InnoDB;
alter table author_books add index FK24C812F63FA913A (book_id),
add constraint FK24C812F63FA913A foreign key (book_id) references book (id);
alter table author_books add index FK24C812F6CD85EDFA (author_id),
add constraint FK24C812F6CD85EDFA foreign key (author_id) references author (id);
```

After running the script you'll have classes similar to these (I've removed empty `mapping` and `constraints` blocks):

```
class Author {
   String name
   static hasMany = [books: Book]
}
```

and

```
class Book {
   String title
   static hasMany = [authors: Author]
   static belongsTo = Author
}
```

`Book` has the line `static belongsTo = Author` because we specified this in `Config.groovy` with the `grails.plugin.reveng.manyToManyBelongsTos` property.

Compos domain class.
The `compos` table has a composite primary key (made up of the `first_name` and `last_name` columns):

```
create table compos (
   first_name varchar(255) not null,
   last_name varchar(255) not null,
   version bigint not null,
   other varchar(255) not null,
   primary key (first_name, last_name)
) ENGINE=InnoDB;
```

and it generates this domain class:

```
import org.apache.commons.lang.builder.EqualsBuilder
import org.apache.commons.lang.builder.HashCodeBuilder
class Compos implements Serializable {
   String firstName
   String lastName
   String other
   int hashCode() {
      def builder = new HashCodeBuilder()
      builder.append firstName
      builder.append lastName
      builder.toHashCode()
   }
   boolean equals(other) {
      if (other == null) return false
      def builder = new EqualsBuilder()
      builder.append firstName, other.firstName
      builder.append lastName, other.lastName
      builder.isEquals()
   }
   static mapping = {
      id composite: ["firstName", "lastName"]
   }
}
```

Since it has a composite primary key, the class is its own primary key so it has to implement `Serializable` and implement `hashCode` and `equals`.

CompoundUnique domain class.
The `compound_unique` table has five properties, three of which are in a compound unique index:

```
create table compound_unique (
   id bigint not null auto_increment,
   version bigint not null,
   prop1 varchar(255) not null,
   prop2 varchar(255) not null,
   prop3 varchar(255) not null,
   prop4 varchar(255) not null,
   prop5 varchar(255) not null,
   primary key (id),
   unique (prop4, prop3, prop2)
) ENGINE=InnoDB;
```

and it generates this domain class:

```
class CompoundUnique {
   String prop1
   String prop2
   String prop3
   String prop4
   String prop5
   static constraints = {
      prop2 unique: ["prop3", "prop4"]
   }
}
```

Library and Visit domain classes.
The `library` and `visit` tables have a one-to-many relationship:

```
create table library (
   id bigint not null auto_increment,
   version bigint not null,
   name varchar(255) not null,
   primary key (id)
) ENGINE=InnoDB;
create table visit (
   id bigint not null auto_increment,
   library_id bigint not null,
   person varchar(255) not null,
   visit_date datetime not null,
   primary key (id)
) ENGINE=InnoDB;
alter table visit add index FK6B04D4B4AEC8BBA (library_id),
add constraint FK6B04D4B4AEC8BBA foreign key (library_id) references library (id);
```

and they generate these domain classes:

```
class Library {
   String name
   static hasMany = [visits: Visit]
}
```

```
class Visit {
   String person
   Date visitDate
   static belongsTo = [library: Library]
   static mapping = {
      version false
   }
}
```

`visit` has no `version` column, so the `Visit` has optimistic lock checking disabled (`version false`).

Other domain class.
The `other` table has a string primary key, and an optimistic locking column that's not named `version`. Since we configured this with the `grails.plugin.reveng.versionColumns` property, the column is resolved correctly:

```
create table other (
   username varchar(255) not null,
   nonstandard_version_name bigint not null,
   primary key (username)
) ENGINE=InnoDB;
```

```
class Other {
   String username
   static mapping = {
      id name: "username", generator: "assigned"
      version "nonstandard_version_name"
   }
}
```

User and Role domain classes.
The `user` and `role` tables have a many-to-many relationship, which uses the `user_role` join table:

```
create table role (
   id bigint not null auto_increment,
   version bigint not null,
   authority varchar(255) not null unique,
   primary key (id)
) ENGINE=InnoDB;
create table user (
   id bigint not null auto_increment,
   version bigint not null,
   account_expired bit not null,
   account_locked bit not null,
   enabled bit not null,
   password varchar(255) not null,
   password_expired bit not null,
   username varchar(255) not null unique,
   primary key (id)
) ENGINE=InnoDB;
create table user_role (
   role_id bigint not null,
   user_id bigint not null,
   date_updated datetime not null,
   primary key (role_id, user_id)
) ENGINE=InnoDB;
alter table user_role add index FK143BF46A52388A1A (role_id),
add constraint FK143BF46A52388A1A foreign key (role_id) references role (id);
alter table user_role add index FK143BF46AF7634DFA (user_id),
add constraint FK143BF46AF7634DFA foreign key (user_id) references user (id);
```

The `user_role` table has an extra column (`date_updated`) and would be ignored by default, but since we configured it with the `grails.plugin.reveng.manyToManyTables` property it's resolved correctly:

```
class Role {
   String authority
   static hasMany = [users: User]
   static belongsTo = User
   static constraints = {
      authority unique: true
   }
}
```

```
class User {
   boolean accountExpired
   boolean accountLocked
   boolean enabled
   String password
   boolean passwordExpired
   String username
   static hasMany = [roles: Role]
   static constraints = {
      username unique: true
   }
}
```

Thing domain class.
The `thing` table has a non-standard primary key column (`thing_id`) and a unique constraint on the `email`

column. The `name` column is nullable, and is defined as `varchar(123)`:

```
create table thing (
   thing_id bigint not null auto_increment,
   version bigint not null,
   email varchar(255) not null unique,
   float_value float not null,
   name varchar(123),
   primary key (thing_id)
) ENGINE=InnoDB;
```

and it generates this domain class:

```
class Thing {
   String email
   float floatValue
   String name
   static mapping = {
      id column: "thing_id"
   }
   static constraints = {
      email unique: true
      name nullable: true, maxSize: 123
   }
}
```

9. Update a table and re-run the script.
Add a new column to the `thing` table:

```
alter table thing add new_column boolean;
```

We'll re-run the script but need to configure it to generate the updated domain class in a different directory from the default so we can compare with the original. To configure this, set the value of the `grails.plugin.reveng.destDir` property in `grails-app/conf/Config.groovy`:

```
grails.plugin.reveng.destDir = 'temp_reverse_engineer'
```

Also change the configuration to only include the `thing` table:

```
grails.plugin.reveng.includeTables = ['thing']
```

Re-eun the reverse-engineer script:

```
grails reverse-engineer
```

The script will generate this domain class in the temp_reverse_engineer/com/revengtest folder:

```
class Thing {
    String email
    float floatValue
    String name
    Boolean newColumn
    static mapping = {
        id column: "thing_id"
    }
    static constraints = {
        email unique: true
        name nullable: true, maxSize: 123
        newColumn nullable: true
    }
}
```

The domain class has a new `Boolean newColumn` field and a `nullable` constraint. Since this generated the correct changes it's safe to move replace the previous domain class with this one.