Liquibase - Open Source Tool for Database Version Control

Nathan Voxland

You would never develop code without version control, why do you develop your dat without it? Liquibase is an open source library for tracking, managing and applying dat changes that can be used for any database with a JDBC driver. It is built on a s premise: all database changes are stored in a human readable yet trackable form checked into source control.

Web Site: http://liquibase.org
Version Tested: 2.0.1
License & Pricing: Apache 2.0 License, Free
Support: Community forum (http://forum.liquibase.org) or commercial support and training from http://liquibase.com

At its simplest form, Liquibase is a tool that reads a list of database changes from a changelog file. The file starts out empty when you begin your application, but as you make changes to your application that require corresponding changes in the underlying database, you append a description of the changes to the changelog file. The description of the changes can be standard SQL commands such as "add column" or more complex descriptions such as "introduce lookup table". The changelog file is often XML-based, but does not have to be. Once a new change is added to the changelog file, Liquibase will read the file, detect the new change, and apply it to the database. When you commit your code changes to your version control system, you commit the changelog file containing the database "version" alongside it.

What you end up with is a mixture of a database version control system and database refactoring tool. As your database evolves over the course of a project, Liquibase ensures that the database you deploy to production has the same schema that the application code expects and has been tested against.

Compared to other database change tracking tools, Liquibase has three defining differences:

1. Understand how the database changed, not just what changed

A common approach to database change management is:

1. Take a database snapshot at the beginning of a project
2. Allow developers to make the changes they need as the develop their code
3. Compare the final database to the original database.
4. From this comparison, auto-generate the SQL to add, remove and modify tables, columns, views, etc.

A prime example of this is Hibernate's hbm2ddl library, which allows you to create your Java to database mapping, then determine the SQL needed to make the database match the schema the code expects. While this approach works for many source code version control systems, there is an inherent problem applying it to databases: the **way** you get from the start to the final state is as important as getting there. A simple example is a change that renames the person.fname column to person.first_name. A database comparison would see that there is no longer a person.fname column and there is now a person.first_name column. The SQL it would generate would be:

```
alter table person drop column fname;
alter table person add column first_name varchar(255);
```

While the resulting database schema is the correct form, when you apply the above SQL to your production database, you will find that all your records have null first_names. Because the tool was not smart enough to understand **how** the schema changed, you have lost data.

There are many similar examples of the importance of knowing how the schema changed. In order to protect the database, Liquibase does not use automatic comparison but instead relies on a changelog being built up manually one changeset set at a time. It may sound tedious at first glance, but as you will see, the process is streamlined and in practice it fits well with standard development techniques.

2. Handle Multiple Developers and Multiple Branches/Merges

There are database versioning tools rely on the manual creation of SQL or SQL-like changesets, but many use a simplistic tracking system that does not scale to multiple developers or code branches. In particular, they are built around a concept of a linear

database "version" which starts at version 1. After a change is added, the version is incremented to 2, then 3, etc. When an existing database is set to be updated, the current version is determined and all the changesets after that version are applied.

This works well for projects where only one person adds changesets and/or there are never any branches, but it quickly breaks down when separate developers attempt to add "version 4" to the database concurrently. This can be managed with good communication, but falls apart completely when versions 4 to 10 were added in a feature branch that needs to be integrated back into the master branch that already has version to 20 added.

Liquibase does away with this issue by using a unique identification scheme for each changeset that is designed to guarantee uniqueness across developers and branches while still being easy to manually manage. As you will see in the below examples, each Liquibase changeset contains two attributes: an "id" and an "author". These two attributes along with the name and path of the file make up the changeset identifier Liquibase uses to determine if it has been ran against a given database. At update time, each changeset is compared against the list of applied changesets and it is executed if and only if it has not been before. Since the comparison is done for each changeset instead of being based on a single "version", any new changesets brought into the changelog file - whether from a different developer or from a different branch - will be correctly executed.

3. Higher Level Refactorings

Finally, Liquibase supports not just standard create/alter/update SQL statements, but higher level database "refactorings" such as "split column" and "introduce lookup table" which allow complex database changes to be described and managed easily. This not only makes the initial changelog creation easier, but improves readability and traceability. Furthermore, you gain the ability to support updating and managing the same schema across multiple database vendors using the same changelog file.

Liquibase also supports a powerful extension model which allows you to define arbitrarily complex changes as well as add functionality to the built-in refactorings. This allows you to wrap update patterns that are common to you into an easy to use and manage package.

Using Liquibase

To see how Liquibase works, start with a blank database and a blank changelog file in a directory containing the unzipped Liquibase binary from http://liquibase.org/download. If you name your blank changelog file db.changelog.xml, your directory structure will look like:

```
- lib
o JDBC Driver jar file(s) for your database
- db.changelog.xml
- liquibase
- liquibase.bat
- liquibase.jar
```

Liquibase is built on the Java platform, and therefore requires the Java runtime environment (1.5+) to be installed in your system and the corresponding JDBC driver jar files in the lib folder. Contact your database vendor for information on available drivers and jdbc url structure. If everything is installed correctly, running:

```
>liquibase.bat --version
```

Will output:

```
Liquibase Version 2.0.1
```

Open the blank db.changelog.xml file paste in

```
<databaseChangeLog
xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-2.0.xsd">
</databaseChangeLog>
```

You can now update your database using the db.changelog.xml file by running

```
>liquibase.bat --changeLogFile=db.changelog.xml
--username=USER --password=PWD --url=jdbc:mysql://localhost/liquibase update
```

with correct values for username, password, and url.

You will see the output:

```
INFO 4/14/11 12:08 AM:liquibase: Successfully acquired change log lock
INFO 4/14/11 12:08 AM:liquibase: Creating database history table with name: `DATABASECHANGEL
INFO 4/14/11 12:08 AM:liquibase: Reading from `DATABASECHANGELOG`
INFO 4/14/11 12:08 AM:liquibase: Successfully released change log lock
Liquibase Update Successful
```

If you look at your database, you have two new tables: "databasechangelog" and "databasechangeloglock". These two tables are used by Liquibase to track changes and

execution. There is still nothing else in your database because there is nothing in your db.changelog.xml file.

If you replace insert

```
<changeSet id="1"
author="nvoxland">
        <createTable tableName="person">
                <column name="id" type="int"
                autoIncrement="true">
                <constraints primaryKey="true" nullable="false"/>
                </column>
                <column name="firstname" type="varchar(50)"/>
                <column name="lastname" type="varchar(50)">
                <constraints nullable="false"/>
                </column>
        </createTable>
</changeSet>
```

into the <databaseChangeLog> root element and run the Liquibase.bat update com
list above again, you will see the output:

```
INFO 4/14/11 12:14 AM:liquibase: Successfully acquired change log lock
INFO 4/14/11 12:14 AM:liquibase: Reading from `DATABASECHANGELOG`
INFO 4/14/11 12:14 AM:liquibase: ChangeSet db.changelog.xml::1::nvoxland ran successfully in
INFO 4/14/11 12:14 AM:liquibase: Successfully released change log lock
Liquibase Update Successful
```

and the database now contains a person table.

Building your ChangeLog

The above pattern of:

1. Append new changeSet to databaseChangeLog
2. liquibase.bat update

is repeated over and over as you need new changes to your database. The changeSets can include any of the changes listed at http://liquibase.org/manual/home, raw SQL, or any custom changes you create using the extension framework (http://liquibase.org/extensions). For a common development process with multiple developers, the pattern is extended to:

1. Write code
2. Find you need a database change
3. Append new changeSet to db.changelog.xml
4. liquibase.bat update
5. Test code and database
6. Repeat 1-4 as necessary
7. Update local codebase from version control
8. liquibase.bat update to apply changes from other developers
9. Repeat 1-8 as necessary
10. Commit your code and db.changelog.xml to version control
11. When ready, update QA database with db.changelog.xml built up during development
12. When ready, update production database with db.changelog.xml built up during development

Additional ways to run Liquibase

If the command line interface does not fit your needs, Liquibase can be ran on demand via Ant, Grails or Maven, or can be ran automatically as part of application startup using the built in Spring or Servlet Listener support or interacting with a simple Java façade API.

Managing ChangeLogs

The above example uses a single XML file that contains all the changeSets, but as your project grows, you may want to break changelogs into multiple files using the <include> tag. Depending on your needs, you can create changelog files in:

- XML
- Raw SQL (which is treated as a single changeSet)
- Formatted SQL (which allows you to break it up into multiple changeSets)
- DSL-style format
- Create your own format using the extension framework.

For example, the person example above could be stored in db.changelog.sql and written as:

```
--liquibase formatted sql
--changeset nvoxland:1
CREATE TABLE person (
    id int PRIMARY KEY,
    firstname varchar(255),
    lastname varchar(255) NOT NULL
);
```

or, using the grails plugin from http://grails-plugins.github.com/grails-database-migration/, as

```
databaseChangeLog = {
        changeSet(author: 'nvoxland', id: '1') {
                createTable(tableName: 'person') {
                column(autoIncrement: 'true', name: 'id', type: 'BIGINT') {
                constraints(nullable: 'false', primaryKey: 'true')
                }
                column(name: 'firstname', type: 'VARCHAR(255)')
                column(name: 'lastname', type: 'VARCHAR(255)') {
                        constraints(nullable: 'false')
                }
                }
        }
}
```

Additional Liquibase Features

Beyond tracking and applying changes to a database, Liquibase supports many powerful features including:

- *Rollback Support:* If you want to undo an update, *liquibase.bat rollback* allows you to roll back changeSets based on number of changeSets,to a given date, or to a given tag stored in the database
- *Update/Rollback SQL Output:* rather than executing updates or rollbacks directly against the database, you can generate the SQL that would be ran for inspection and/or manual execution.
- *Future Rollback Output:* Before you apply an update to a database, you can generate the SQL you would need to run in order to bring the database back to the state it is in now for inspection.
- *ChangeLog and ChangeSet preconditions:* Preconditions can be added to the changeLog or individual changeSets to check the state of the database before attempting to execute them
- *DBDoc:* You are able to generate Javadoc style documentation for your current schema and its history. See http://www.liquibase.org/dbdoc/index.html for example output
- *ChangeSet Contexts:* ChangeSets can be assigned "contexts" in which to run. Contexts are selected at runtime and can be used to have changeSets that only run in test instances or other unique circumstances
- *ChangeSet checksums:* When a changeSet is executed, Liquibase stores a checksum and can fail or alter execution if it detects a change between the original definition of a changeSet when it was run and the current definition.
- *Diff Support:* Although Liquibase is built to use database comparisons for change management, there is support for it in Liquibase which is helpful in many cases such as performing sanity checks between databases.

More Information

For more information, visit http://liquibase.org. There you will find documentation, videos, downloads, and more.

More Database Content

- Behavior Driven Database Development (BDDD)
- Database Videos and Tutorials

Click here to view the complete list of tools reviews

This article was originally published in the Summer 2011 issue of Methods & Tools