Liquibase

Database change management

Sergii Fesenko

Software development

- We do a lot of changes during development
- We use version control system for tracking code changes
- However, tracking DB changes is not as easy as tracking code changes

Database development

There are many different artifacts in database:

- tables
- data, that lives in the tables
- stored procedures
- views
- triggers
- etc (DBMS specific artifacts)

What is wrong with DB?

- database has persistent state (that’s why we use it)
- code and database must corresponds
- each developer has local environment
- we change code very often (we agile :)

Actually, these issues are Agile specific, in case of waterfall they disappears :)

What is solution?

Store DB changes in text files (DDL and DML) and apply them manually to DB.

A lot of projects use this approach

Manual tracking DB changes

Possible issues:

- easy to lost sync between code and DB state
- hard to recover from error during development (ex: in case of applying wrong statement to DB)
- often require to re-create DB from scratch during development
- hard to find out state of particular DB environment
- may require to support 2 versions of changes: full and incremental

What tools are available?

- Flyway
- Liquibase
- c5-db-migration
- dbdeploy
- mybatis
- MIGRATEdb
- migrate4j
- dbmaintain
- AutoPatch

What makes liquibase different?

- DDL abstraction DSL
- Support for different changelog format
  - build-in: XML, YAML, JSON and SQL
  - community-managed: groovy and closure
- Over 30 built-in database refactorings
- Rollback database changes feature
- Database diff report
- Extensibility

Liquibase major concepts

- Changelog file
- Changeset
- Changes
- Preconditions
- Contexts

Liquibase changelog file

The root of all Liquibase changes is the databaseChangeLog file.

```
<databaseChangeLog>
  <changeSet id="1" ...>
            ...
  </changeSet>
  <changeSet id="2" ...>
            ...
  </changeSet>
</databaseChangeLog>
```

Liquibase changelog file

It is possible to break up changelogs into multiple manageable pieces

```
<databaseChangeLog>
    <include file="src/db/tables.xml"/>

    <include file="src/db/views.xml"/>

    <include file="src/db/data.xml"/>
                ...
</databaseChangeLog>
```

Liquibase changeset

Changeset is a group of changes

Liquibase attempts to execute each changeSet in a transaction that is committed at the end, or rolled back if there is an error.

Liquibase changeset

```
<databaseChangeLog>

<!-- id, author and current file name must be unique -->
  <changeSet id="1" author=”sfesenko” >
        <comment>Sample of changeset</comment> <!-- comment is optional -->

    <change .. /> <!-- will be explained later -->

<!-- rollback is a changeset that must be executed in order to “cancel” current changeset --
>
    <rollback>
        <change .. />
    </rollback>
  </changeSet>
</databaseChangeLog>
```

Liquibase changeset

```
<changeSet id="1" author=”sfesenko” runAlways="true"> <!-- false by default -->
 …
</changeset>

<changeSet id="2" author=”sfesenko” runOnChange="true"> <!-- false by default -->
 …
</changeset>
```

Useful for views, triggers, stored procedures

```
<changeSet id="3" author=”sfesenko” failOnError="false"> <!-- true by default -->
 …
</changeset>
```

Liquibase changeset preconditions

Both changelog and changeset may have **precondition** - assertion, that will be evaluated before execution of changeset.

Available preconditions:

- check for dbms type
- check for current user name
- check if changeset has been executed
- check if table exists
- check if table has column
- check if view exists
- check if FK constraint exists
- check if index exists
- check if sequence exists
- check if table has primary key (or specified primary key exists)
- arbitrary sql check

Liquibase changeset preconditions

Sample of precondition usage:

```
<changeSet id="1" author=”sfesenko” >
        <preConditions onFail="MARK_RAN">
            <tableExists tableName="TEST" />
       </preConditions>
            <<dropTable cascadeConstraints="true" tableName=”TEST” />
</changeset>
```

Liquibase changeset context

It’s possible to specify for each changeset in what context it should be run.
Context value can be specified on liquibase run.

```
<changeSet id="1" author=”sfesenko” context=”clear”>
        <delete tableName="USERS" />
</changeset>
```

Liquibase change

Each changeset contains change which describes required modification of database.

It’s possible to use DSL (recommended, dbms independent) or SQL for defining a change.

Liquibase change

```
<changeSet id="1" author="sfesenko">
        <createTable tableName="CURRENCY">
                <column name="CURRENCY" type="varchar(12)" >
                                        <constraints primaryKey="true"
primaryKeyName="PK_CURRENCY_CURRENCY"/>
                                </column>
                                <column name="ISO" type="java.lang.Integer" />
                                <column name="DECIMAL_RULES" type="varchar(12)" />
                  </createTable>

                  <sql>
                CREATE TABLE CURRENCY (
                                        CURRENCY                VARCHAR2(12) NOT NULL,
                                        ISO                     NUMBER(10),
                                        DECIMAL_RULES     VARCHAR2(12),
                                        CONSTRAINT PK_CURRENCY_CURRENCY PRIMARY KEY (CURRENCY)
                                );
                  </sql>
    </changeSet>
```

Liquibase changes/refactorings

Structural Refactorings

- Add Column
- Rename Column
- Modify Column
- Drop Column
- Alter Sequence
- Create Table
- Rename Table
- Drop Table
- Create View
- Rename View
- Drop View
- Merge Columns
- Create Stored Procedure

Data Quality Refactorings

- Add Lookup Table
- Add Not-Null Constraint
- Remove Not-Null Constraint
- Add Unique Constraint
- Drop Unique Constraint
- Create Sequence
- Drop Sequence
- Add Auto-Increment
- Add Default Value
- Drop Default Value

Referential Integrity Refactorings

- Add Foreign Key Constraint
- Drop Foreign Key Constraint
- Drop All Foreign Key Constraints
- Add Primary Key Constraint
- Drop Primary Key Constraint

Non-Refactoring Transformations

- Insert Data
- Load Data
- Load Update Data
- Update Data
- Delete Data

Architectural Refactorings

- Create Index
- Drop Index
- Custom Refactorings

Modifying Generated SQL

- Custom SQL
- Custom SQL File
- Custom Refactoring Class
- Execute Shell Command

How to run liquibase

On Demand

- Command Line
- Ant
- Maven

Automated

- Servlet Listener
- Spring Listener
- JEE CDI Listener

Update DB on application start

Java APIs

Liquibase can easily be embedded and executed through its Java APIs.

Liquibase command line

1. Create property file liquibase.property with connection parameters:

```
driver: oracle.jdbc.OracleDriver
classpath: ojdbc14-10.2.0.3.0.jar
url: jdbc:oracle:thin:@localhost:1521:oracle
username: SQ_BANK
password: SQ_BANK
```

2. Run liquibase as

./liquibase <command>

Liquibase quickstart

1. Generate changelog from existing database

Run ./liquibase --changeLogFile=changelog.xml generateChangeLog

1. Dump data to xml file

Run ./liquibase --changeLogFile=data.xml --diffTypes=data generateChangeLog

Be ready for OutOfMemoryError

Liquibase commands

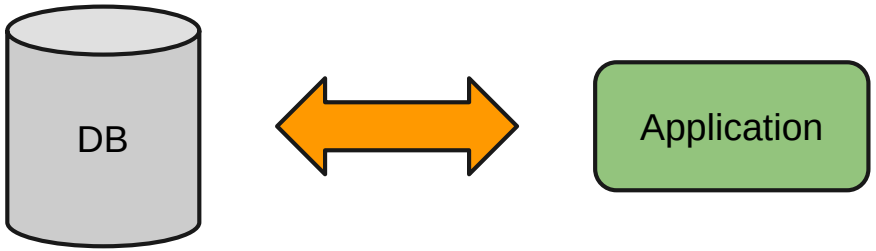
- update
- updateSQL
- validate
- status
- dropAll
- rollbackCount
- generateChangeLog
- diff
- changelogSync
  - changelogSyncSQL
- clearCheckSums

Liquibase tips and tricks

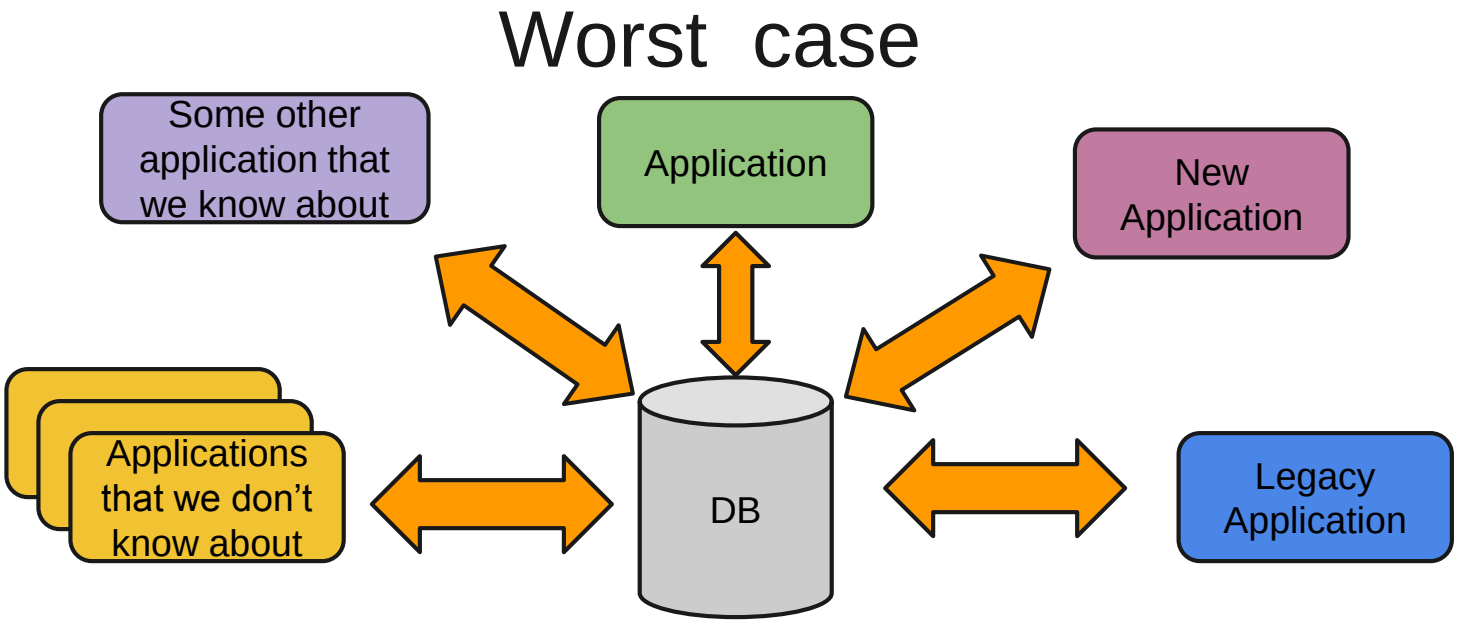
- Use one file per changeset (greatly simplified merges)
- Use convention for file names
- Use “run on change” attribute for stored procedures, views, triggers, etc
- Decide early if rollback feature will be used
- Test both incremental and full update
- Do not change “wrong” changeset - just add new one with fix
- Consider possibility of “compressing” changesets when full update became slow

When it is good

Best case



When it is not so good



Q & A