Liquibase

Installation in Windows

1. Go to <https://www.liquibase.org/download>
2. Provide your email address and you will be redirected to Liquibase Download Page
3. Click on Installer for Windows
4. A .exe installer file will be downloaded in your desktop.
5. Run it to get liquibase installed in Windows machine
6. To check if installation was successful, Go to Command Prompt and type
   1. liquibase –version

Getting Started with Liquibase

1. Create a folder where you want to start your project
2. Create a sub directory to add connectors in it and name it as Connector
3. Create a properties file and name it as liquibase.properties
4. Create a SQL changelog file

Tracking Tables

There are two types of tracking tables in liquibase:

1. DATABASECHANGELOGs
2. DATABASECHANGELOGLOCKs
3. DATABASECHANGELOG Table

This table is created and used by liquibase to keep track of changesets which has already been run.

It does this by tracking each changeset as a row, identified bby a combination of id, author and filename specified in the changelog.

1. DATABASECHANGELOGLOCK Table

This table is used by liquibase to prevent multiple instances of liquibase running at one time.

If liquibase doesnot exit quickly, the lock row might get stuck in locked state preventing others from making database changes. If thiw happens, we can clear out the current lock by running:

liquibase releaseLocks

which runs

UPDATE DATABASECHANGELOGLOCK SET LOCKED=0

Developer Workflow

Step 1: Adding changeset to changelog

Step 2: Verifying SQL we are excuting

Step 3: Saving your changelog to source code

Step 4: Running database update command

Step 5: Verifying that changesets are executed

Step 1: Adding changeset to changelog

Step 2: Verify changes we are going to make

Liquibase contains number of helper commands that will generate SQL query that we are going to execute.

We can use updateSQL command to inspect SQL liquibase will run while using the update command.

The command is run as:

Liquibase –changeLogFile=file\_name.<db\_type>.sql updateSQL

Liquibase –changeLogFile=file\_name.xml updateSQL

We can now verify SQL query generated and if there are unexpected changes, review the changelog, edit, then run the command again if needed.

Step 3: Save your changelog to source control

Only we verify changes, save your changelog to source control such as Githu, Bitbucket, etc.

Step 4: Running the liquibase command update

Run update command as:

Liquibase –changeLogFile=file\_name.<db\_type>.sql update

Liquibase –changeLogFile=file\_name.xml update

Step 5: Verify that your changes were executed

Different ways to do this are:

1. Run liquibase history command to see list of deployed changes
2. Run liquibae status command to see list of undeployed changes
3. Run diff command between two database to see if there are missing or unexpected changes detected.
4. Open you database IDE and dircetly inspect the databbase.

Most common commands used in liquibase

1. generateChangeLog
2. update
3. snapshot
4. diff
5. diffChangeLog
6. Rollback commands
7. Targeted rollback commands

generateChangeLog

This command is used to create a changelog file that has a sequence of changesets which describes to liquibase how to re-create the current state of changesets which describes to liquibase how to re-create the current state of the source database.

It is used when there is existing dataase bbut liquibase has not bbeen used efore.

Run this command as:

Liquiase –changeLogFile=file\_name.mysql.sql generateChangeLog

Update

This command is used to apply database changes that we have aded to our changelog.

When we run the update command, liquibase will read the changesets in changelof file in sequential order than compares the unique identifiers of id, author and path to finename to the values stored in the DATABASECHANGELOG table.

1. If the unique identifiers do not exists, liquibase will apply the changeset to the database.
2. If the unique identifiers do exist, MD5Sum of the changeset is compared to one in the dataase.
3. If they are different, liquibae will either produce an alert error message that someone has changed it unexpectedly, or will re-apply the changeset if the status of the runOnChange changeset attribute is set to TRUE.

Two ways to run update command:

1. liquibase --changeLogFile=**mychangelog.<db\_type>.sql** update
2. liquibase --changeLogFile=**mychangelog.<db\_type>.sql** updateCount 10

snapshot command

It is used when we want to quickly compare changes in our database or keep a record of our current database schema.

It is similar to a photograph of our database that can be used in diff command or diffChangeLog command.

Uses:

The snapshot command has two modes:

1. When run without options, it will gather the current state of the database and show a text-based version of the schema to STDOUT.
2. When run with the –snapshotFormat=yaml option, it creates a yaml-based output that represents the current state of our dataase.

Snapshots can e used to compare:

1. Aprevious database state to an online database
2. Aprevious database state to another snapshot

Running diff command with at least one snapshot.yaml file is faster than using a diff command with two online databases.

**liquibase --outputFile=myschemaSnapshot.yaml snapshot --snapshotFormat=yaml**

**liquibase --outputFile=myschemaSnapshot.json snapshot --snapshotFormat=json**

**Liquibase Community snapshot categories:**

* Catalog
* Column
* Foreign Key
* Index
* Primary Key
* Schema
* Sequence
* Unique Constraints
* View

Diff command

Compare two dataase schemas of same type or different type

We need to add referenceUrl and other reference credentials in liquibase.properties file which is considered as source.

There are three main reasons to run this command:

1. To find missing objects etween one database and another.
2. To see that a specific change was made to the database.
3. To find unexpected items in the database.

Categories and Descriptions:

Descriptions

1. Missing: Objects are present on the source database (referenceURL) that are not on the target database (URL)
2. Unexpected: Ojects are present on the target database (URL) is different than as it exists on the source databas (rereferenceURL).

### Categories:

* Catalog
* Column
* Foreign Key
* Index
* Primary Key
* Schema
* Sequence
* Procedure
* Unique Constraints
* View

Liquibase allows us to use diffType attributes to filter types of objects to compare.

Multiple filters can be added as a comma-separated list.

Example: **liquibase --diffTypes=tables, indexes, views diff**

Run this command as :

Liquibase –outputFile=mydiff.txt diff

Liquibase –outputFile=mydiff.json diff –format=json (only for pro)

diffChangeLog command

It creates a changelog file containing deployale changesets, but liquiase determines what these changesets should be based on a comparison of two specified databases.

Uses of diffChangeLog command

This command will provide information in three main areas:

1. To find missing objects in the database
2. To see changes made to the database
3. To find unexpected items in the database

Run this command as :

**liquibase  --changeLogFile=file\_name.mysql.sql diffChangeLog**

The diffChangeLog will be created in XML, YAML, JSON or SQL depending on the changeLofFile attribute set in the liquibase.properties file or CLI.

Rollback commands

rollbackDate date/time

rollback all changes from now to date specified

Run this commnad as:

Liquibase rollbackDate 2020-01-01

rollbackCount <num>

rollbback specified number of changesets

Run this command as:

Liquibase rollackCount 2

rollback tag

reverts all changes made to the dataase ased on the specifie tag.

Run this command as:

Rollack <tag\_name>

If we have not set any tags, run liquibase tag tag\_name to specify tag name for any of our releases.

Targeted rollback commands (pro)

rollbackOneChangeSet command

undo one change

Run helper command as:

**liquibase rollbackOneChangeSetSQL --changeSetAuthor="Liquibase Pro User" --changeSetID="1::createProc-proschema" --changeSetPath=changelog.xml**

Run command as:

**liquibase rollbackOneChangeSet --changeSetAuthor="Liquibase Pro user" --changeSetID="1::createProc-proschema" --changeSetPath=changelog.xml --force**

### The force parameter at the end of the syntax indicates the intention to run the command.

rollbbackOneUpdate

revert all changes of a deploeyement id

Run helper command as:

* **liquibase rollbackOneUpdateSQL --deploymentID=<somedeploymentID>**

Run command as:

* **liquibase rollbackOneUpdate --deploymentID=<somedeploymentID> --force**

Best practices

1. Organize your changelogs

 It is a best practice to create a directory in your classpath to store your *changelogs,* preferably near your database access classes. **You then call all these *changelogs* from a "master" *changelog*, which itself is always an .xml formatted file.**

1. Add comments to changesets

This will save our time in long run.

* Single line comments in SQL require two hyphens: **-- my comment**
* Multi-line comments in SQL use slashes around an asterisk: **/\* my comment \*/**
* Comments in XML use the following format: **<!-- my comment -->**

1. Use <include> tag
2. Use <includeAll> tag
3. Trim your changelogs

### Use Case #1

Suppose you have one *changeset* that creates a **cart table** and a different *changeset* that deletes it. To trim your *changelog*, remove both *changesets* from your *changelog* file but make sure there are no *changesets* between the create and delete that make use of the table, or they will fail on a fresh database build.

### Use Case #2

You have a **cart** **table** that is created in one *changeset*, then a **promo\_code** column is created in another, and an **abandoned** **flag** is created in a third *changeset*.

One option is to combine everything into a **new *changeset*** using the existing **id="1"** and **delete** the other *changesets*.

In <path where liquibase is installed>/examples/sql, sample of liquiase.properties file will be present.

liquibase.properties file

Sample liquibase.properties file

url=jdbc:mysql://localhost:3306/test?autoReconnect=true&amp;useSSL=false

username=root

password=root

classpath=D:/Sony/Learn/Liquibase/Connector/mysql-connector-java-5.1.48-bin.jar

Creating a changelog file

Sample changelog.sql file

-- liquibase formatted sql

-- changeset Sony:1

create table emp(id int not null primary key auto\_increment, emp\_name varchar(100));

Using validate command

Validate command checks and identifies any possible errors in a changelog that can cause the command to fail.

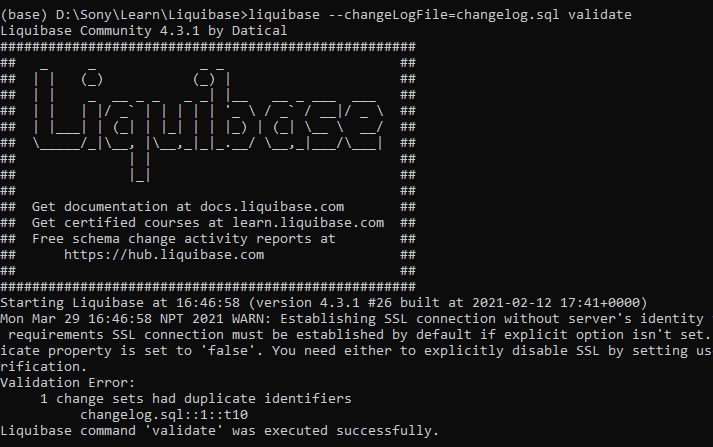
It is used to detect if there are any issues with a changelog before running the update command. With the help ppf validate command, we can avoid a partial update.

Running the validate command

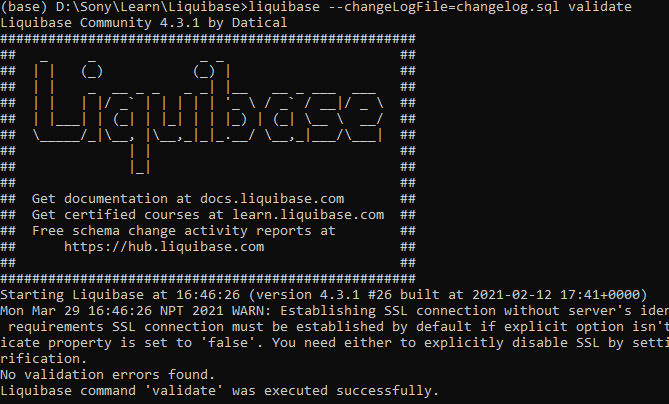
* Configure the liquibase.properties file to include the:
  1. jdbc driver
  2. classpath
  3. database URL
* Open your CLI and run the following command:

liquibase --changeLogFile=mydbchangelog.xml validate

Error detected using validate command



Successful execution of validate command



Why shoul we validate rollbacks

Validating rollbacks

**A complete test cycle should include:**

1. Deploying all changes to the database and validating that they were deployed.
2. Rolling back all changes to the database and validating that all changes were undone and the database was brought back to the previous state.
3. Redeploying all changes to the database.

Auto-generated vs Custom Rollback SQL

Liquibase allows us to undo changes we have made to our database, either automatically or via custom rollback SQL.

Auto-generated rollbacks

Liquibase can automatically generate a rollback statement when it can determine, without a doubt, how the rollback should be constructed. For example, the rollback of a **CREATE TABLE** would be a **DROP TABLE**. If your *changelog* only contains statements that fit into this category, your rollback commands will be generated automatically. You do not need to add anything to your *changelogs* or Liquibase configuration to enable auto-generated rollbacks.

Custom Rollback SQL

Example of Custom Rollback SQL

-- liquibase formatted sql

-- changeset lbuser:20200908

insert into CustomerDetails values ('A','Customer NumberOne', 'Austin')

insert into CustomerDetails values ('A','Customer NumberTwo', 'Dallas')

-- rollback delete from CustomerDetails where id='A'

Advanced Concepts

1. Changeset

A *changeset* is a unit of change that Liquibase can execute on a database. A list of changes created by *changesets* is tracked in a *changelog*. A *changeset* is defined by both an **author** and an **id**attribute (**author:id)**, as well as the *changelog* file path. Both **author** and **id** must be included to execute the *changeset*.

* It is best practice to execute one change per *changeset*using the writer's name as the author.  Using **unknown** as author will work but makes it difficult to know who made the change.
* When Liquibase is run, it will execute your *changesets* in order and check them against the **DATABASECHANGELOG** table to prevent duplicate changes by comparing the *changeset ID, author, and file path*.
* If Liquibase determines a *changeset* has already run, it will skip it unless directed to execute with a "**runAlways**" tag set to "true" in the *changeset*. The **id** tag does not affect the order of *changeset*execution.

updateTestingRollback command

updateTestingRollback is used to validate rollbacks.

**The following illustrates the order of operations based on a *changelog* with 3 *changesets* ready to deploy:**

1. update changeset1, update changeset2, update changeset3
2. rollback changeset3, rollback changeset2, rollback changeset1
3. update changeset1, update changeset2, update changeset3

futureRollbackSQL Command

The futureRollbackSQL command produces the raw SQL Liquibase would use to revert changes associated with **undeployed** *changesets*. It does not deploy any changes to the database. The futureRollbackSQL command is also useful when auditors need to verify that all database changes have a rollback.

## List of common attributes used with *changesets*:

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| stripComments | Set to true to remove any comments in the SQL before executing, otherwise false. Defaults to true if not set. |
| splitStatements | Set to false to not have Liquibase split statements on ;'s and GO's. Defaults to true if not set. |
| rollbackSplitStatements | Same as splitStatements but for rollback SQL. |
| endDelimiter | Delimiter to apply to the end of the statement. Defaults to ";", may be set to "". |
| rollbackEndDelimiter | Same as endDelimiter but for rollback SQL. |
| runAlways | Executes the *changeset* on every run, even if it has been run before. |
| runOnChange | Executes the change the first time it is seen and each time the  *changeset* has been changed. |
| context | Executes the change if the particular context was passed at runtime.  Any string can be used for the context name and they are checked case-insensitively. |
| logicalFilePath | Use to override the file name and path when creating the unique identifier of *changesets*. Required when moving or renaming *changelogs*. |
| labels | Labels are a general-purpose way to categorize *changesets* like contexts,  but working in the opposite way. Instead of defining a set of contexts  at runtime and then a match expression in the *changeset*, you define  a set of labels in the context and a match expression at runtime. |
| runlnTransaction | Should the *changeset* be run as a single transaction (if possible)?  Defaults to true. **Warning: be careful with this attribute. If set to false and an**  **error occurs partway through running a *changeset* containing**  **multiple statements, the Liquibase DATABASECHANGELOG**  **table will be left in an invalid state.** |
| failOnError | Should the migration fail if an error occurs while executing the *changeset*? |
| dbms | The type of a database which that *changeset* is to be used for.  When the migration step is running, it checks the database type  against this attribute. Valid database type names are listed on the  [supported databases page](https://www.liquibase.org/get-started/databases?_ga=2.58925295.354927648.1590390023-712262013.1588748986). It is possible to list multiple databases  separated by commas. You can also specify that a *changeset* is  **not** applicable to a particular database type by prefixing with !. The keywords all and none are also available. |

This can be aded in changesets to control which changeset will run on particular migration run.

Any string can be used for context name and they are checked case-insensitively.

When we run the migrator through any of the available methods, we can pass in a set of contexts to run. Only contexts marked with the passed contexts will be run.

If we don’t assign a context to a changeset, it will run all the time, regardless of contexts we pass in the migrator.

Contexts can be specifie using AND, OR, ! and parenthesis.

Without parenthesis, the ordder of contexts are !, AND and then OR.

“,” can also be used to separate contexts. This works like an OR operation but with the highest precedence.

**Examples:**

* context=''!test''
* context=''v1.0 or map''
* context=''v1.0 or map''
* context=''!qa and !master''

In changelogfile.sql

--changeset sony:1 context:test

create table t1 (

    id int primary key,

    name varchar(255)

);

With executing migrator

liquibase --changeLogFile=changelog.sql --contexts="test" update

What this migrator does is, it will execute all changesets with context equals to test. If context is not provided in changeset, migrator will be executed in that changeset as well.

endDelimiter attribute

The endDelimiter attribute can be set in a hangelog file to override the default value of ;

It can also be set in liquibase.properties file by setting delimiter=<string>

In changeset, we can specify endDelimiter value as

--changeset Sony:1 endDelimiter:/

Sample example of changelog.sql using endDelimiter attribute

--liquibase formatted sql

--changeset your.name:101 endDelimiter:/

CREATE FUNCTION calcProfit2(cost FLOAT, price FLOAT)

RETURNS DECIMAL(9,2)

DETERMINISTIC

BEGIN

DECLARE profit DECIMAL(9,2);

SET profit = price-cost;

RETURN profit;

END

/

CREATE FUNCTION calcProfit3(cost FLOAT, price FLOAT) RETURNS DECIMAL(9,3)

DETERMINISTIC

BEGIN

DECLARE profit DECIMAL(9,3);

SET profit = price-cost;

RETURN profit;

END

/

<include> tag

The <include> tag is used with master changelog to reference other changelogs.

How to use <include> tag

1. Create a master changelog file.
2. Master changelog file must be in XML, YAML or JSON formats only.
3. Add <include> tag and file references to master changelog.
4. Reference change log files can be in XML, YAML, JSON or SQL formats.

Sample of masterchangelog.sql

<?xml version="1.0" encoding="UTF-8"?>

<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-3.8.xsd">

    <include file="changelog.sql"/>

    <include file="changelog2.sql"/>

</databaseChangeLog>

Sample of two reference changelogfile.sql

changelog.sql

--liquibase formatted sql

--changeset t9:1

create table t9 (

    id int primary key,

    name varchar(255)

);

--rollback drop table t7;

--changeset t9:2

insert into t9 (id, name) values (3, 'name 1');

insert into t9 (id,  name) values (4, 'name 2');

changelog2.sql

--liquibase formatted sql

--changeset t8:1

create table t8 (

    id int primary key,

    name varchar(255)

);

--rollback drop table t8;

--changeset t8:2

insert into t8 (id, name) values (3, 'name 1');

insert into t8 (id,  name) values (4, 'name 2');

Command used

liquibase --changeLogFile=master.xml --contexts="test" update

<includeAll> tag

The <includeAll> tag allows us to specify a directory that contains multiple changelog files.

It is used within aster changelog file to call on the directory and include all changlelog files as indiviual changes.

How to use <includeAll> tag

1. Create a master changelog file.
2. Master changelog file must be in XML, YAML or JSON formats only.
3. Add <includeAll> tag and directory references to master changelog.
4. Change log files inside reference directory can be in XML, YAML, JSON or SQL formats.

Sample of masterchangelog.sql

<?xml version="1.0" encoding="UTF-8"?>

<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-3.8.xsd">

<includeAll path="FirstSet"/>

    <includeAll path="SecondSet"/>

</databaseChangeLog>

Sample of two reference changelogfile.sql

changelog.sql inside FirstSet Directory

--liquibase formatted sql

--changeset t9:1

create table t9 (

    id int primary key,

    name varchar(255)

);

--rollback drop table t7;

--changeset t9:2

insert into t9 (id, name) values (3, 'name 1');

insert into t9 (id,  name) values (4, 'name 2');

changelog2.sql inside SecondSet Directory

--liquibase formatted sql

--changeset t8:1

create table t8 (

    id int primary key,

    name varchar(255)

);

--rollback drop table t8;

--changeset t8:2

insert into t8 (id, name) values (3, 'name 1');

insert into t8 (id,  name) values (4, 'name 2');

Command used

liquibase --changeLogFile=master.xml --contexts="test" update

<labels> tag

Labels are tags that can add to changesets to control which changeset will be executed in any migration run.

Any string can be used for the label name and it is case sensitive.

Sample of changeset test.sql

--liquibase formatted sql

--changeset t12:1 labels:1.0

create table t12 (

    id int primary key,

    name varchar(255)

);

liquibase --changeLogFile=test.sql --labels="1.0" update

Same as contexts

If labels is not defined in changeset but used in migration run command, that particular changeset will get executed.

Labels can be specified using AND, OR, ! and () (paranthesis for grouping)

logicalFilePath Attribute

It is used to override the file name and path when creating the unique identifier of changesets.

It is reuired qhen moving or renaming changelogs to prevent liquibase from redeploying the corresponding changesetd as their unique identifer will have changed by the move.

Sample changelog.sql

--changeset your.name:1 logicalFilePath:path-independent

create table company (

id int primary key,

name varchar(50) not null,

address1 varchar(50),

address2 varchar(50),

city varchar(30)

)

logLevel parameter

The logLevel parameter controls the amount of messages that are generated when running liquibase commands.

It can be applied to all Liquibase commands.

The default value of logLevel parameter is severe.

It can have following values:

1. Off
2. Severe: provides messages regardning serious errors.
3. Warning: helps to resolve errors
4. Info
5. Debug

Examples:

liquibase –logLevel=debug update

Preconditions

Preconditions are changelog or changeset tags which control the execution of as update on the status of the database.

Preconditions are checked at the beginning of the execution of a particular changelog.

--liquibase formatted sql

--changeset Liquibase User:1

--precondition-sql-check expectedResult:0 SELECT COUNT(\*) FROM primary\_table

--comment: /\*comments should go after preCondition. If they are located before the precondition, then Liquibase usually gives error.\*/

create table primary\_table (

id int primary key,

name varchar(50) not null,

address1 varchar(50),

address2 varchar(50),

city varchar(30)

)

Liquibase defines two types of preconditions:

1. Precondition failures which represent that the check failed
2. Precondition erros that are the exceptions thrown in the exeution of a check

The process of both can be controlled through onFail and onError attributes on preConditions tag.

Available attributes

| **Attribute** | **Description** |
| --- | --- |
| onFail | Controls what happens if the preconditions check fails. |
| onError | Controls what happens if there is an error checking whether the precondition passed or not. |
| onSqlOutput | Controls what to do in the updateSQL mode. **Since 1.9.5** |
| onFailMessage | Provides a custom message to output when preconditions fail. **Since 2.0** |
| onErrorMessage | Provides a custom message to output when preconditions fail. **Since 2.0** |

onFail/onError Values

| **Value** | **Description** |
| --- | --- |
| HALT | Halts the execution of the entire *changelog* (default). HALT can be put outside a *changeset* (e.g. at the beginning of the *changelog*). |
| CONTINUE | Skips over the *changeset*. Execution of the *changeset* will be attempted again on the next update. Continues with the *changelog*. |
| MARK\_RAN | Skips over the *changeset* but mark it as executed. Continues with the *changelog*. |
| WARN | Sends a warning and continues executing the *changeset* / *changelog* as normal. WARN can be put outside a *changeset* (e.g. at the beginning of the *changelog*). |

onSqlOutput values

| **Value** | **Description** |
| --- | --- |
| TEST | Runs the *changeset* in the updateSQL mode. |
| FAIL | Fails the preCondition in the updateSQL mode. |
| IGNORE | Ignores the preCondition in the updateSQL mode (default). |

runOnChange changeset attribute

It executes the change the first time it is seend and each time the changeset is modified.

Liquibase determines that a changeset has been modified by comparing the MD5 checksu for the changeset to the checksum stored in DATABASECHANGELOG table.

If the runOnChange attribute is not set or set to false, Liquibase will generate a checksum error if a *changeset* is modified after it has been deployed to a database. This is done to notify you that a *changeset* has been unexpectedly modified.

Example

--changeset your.name:changeset1 runOnChange:”true”

CREATE or REPLACE . .

Database update commands

Update command

Update command deploys any changes that are in the changelog file and that have not been deployed to your database yet.

It is used to apply database changes that are specified in the changelog file to our database.

When we run the update command, liquibase sequentially reads changesets in the changelog file, then compares uniqu identifiers of id, author and file path to the values stored in the DATBASECHANGELOG table.

* If the unique identifiers do not exist, Liquibase will apply the *changeset* to the database.
* If the unique identifiers exist, the MD5Sum of the *changeset* is compared to the one in the database.
* If they are different, Liquibase will produce an error message that someone has changed it unexpectedly. However, if the status of the runOnChange or runAlways *changeset* attribute is set to TRUE, Liquibase will re-apply the *changeset*.

Running the update command

To run the update command, you need to specify your driver, class path, and URL in your liquibase.properties file.

Run the command as:

liquibase --changeLogFile=dbchangelog.xml update

Update command attributes

|  |  |  |
| --- | --- | --- |
| --labels | Tags you can add to *changesets* to determine which *changesets* in the *changelog* to evaluate based on their labels | Optional |
| --contexts | Expressions you can add to *changesets* to determine which *changesets* in the *changelog* to evaluate based on their contexts | Optional |
| --liquibaseSchemaName | Specifies in what Schema the Liquibase DATABASECHANGELOG and DATABASECHANGELOGLOCK tables will be managed | Optional |

liquibase --changeLogFile=test.sql --liquibaseSchemaName=master\_data update

updateSQL

The updateSQL command is a helper command that allows you to inspect the SQL liquibase will run while using the update command.

It is used when we want to inspect the raw SQL before running the update command, so that we can correct any issues that arise before running the command.

liquibase --changeLogFile=test.sql --outputFile=file\_name.sql updateSQL

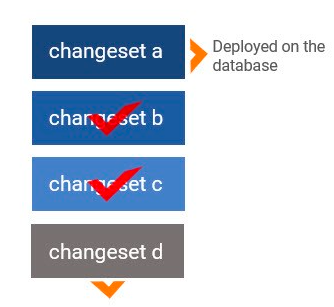
--outputFile specifies the file path to where the update SQL will be written

If this attribute is not provided, output will be displayed in console (STDOUT).

updateCount command

The updateCount <value> command updates a specified number of changestes, where <value> is the number of changesets you want to update sequentially on your database.

The image below shows four changestes: a,b,c,d. If a has already been deployed on the database, so running command updateCount 2, deploys changeset b and c without applying changeset d.



liquibase --changeLogFile=test.sql updateCount 2

updateCountSQL <value> command

The updateCountSQL <value> command is a helper command that inspects the SQL liquibase will run while using the updateCount <value> command.

liquibase --changeLogFile=test.sql --outputFile=file\_name.sql updateCountSQL 2

If outputFile attribute is not provided, output will be displayed in console (STDOUT).

updateTestingRollback command

This command tests rollback support by deploying all pending changesets to the database, executes a rollback seuqnetially for the equal number of changesets that were deployed, and then runs the update again deploying all changesets to the database.

updateTestingRollback is typically used when you want to test rollback functionality when deploying *changesets* in your *changelog* sequentially.

updateTestingRollback utilizes a multi-step operation and runs in sequential order:

1. update changeset1; update changeset2; update changeset3
2. rollback changeset3; rollback changeset2; rollback changeset1
3. update changeset1; update changeset2 update changeset3

Run this command as:

liquibase --changeLogFile=test.sql updateTestingRollback

Diff commands

generateChangeLog command

The generateChangeLog command creates a changelog file that has a sequence of changesets which describe how to re-create the current state of the database.

It is used when we want to capture the current state of a database, then apply those changes to any number of database. This is typically only done when a project has an existing database, but hasn’t used Liquibase before.

Run this command using

liquibase --changeLogFile=dbchangelog.xml generateChangeLog

To create SQL changelog file, we need to add database type name when specifying the changelog file as:

liquibase --changeLogFile=expenses2.mysql.sql --logLevel=debug generateChangeLog

diff command

The diff command in Liquibase allows us to compare two databases of same type or different types, to one another.

The diff command is typically used at the completion of a project to verify all expected changes are in the *changelog* or to detect drift between a model schema and a database's actual schema.

The diff command is also useful for the following tasks:

* Finding missing objects between one database and another
* Seeing that a change was made to your database
* Finding unexpected items in your database

Running the diff command

Running the diff command requires two URLs.

referenceURL- the source for comparison.

url- target of the comparison.

Sample of liquibase.properties file

url=jdbc:mysql://localhost:3306/qa?autoReconnect=true&amp;useSSL=false

username=root

password=root

classpath=D:/Sony/Learn/Liquibase/Connector/mysql-connector-java-5.1.48-bin.jar

referenceUrl=jdbc:mysql://localhost:3306/db?autoReconnect=true&amp;useSSL=false

referenceUsername=root

referencePassword=root

liquibase --outputFile=db\_vs\_qa.txt diff

The diff command produces a list of categories along with one of the following description:

1. Missing: there are objects on our source database (referenceUrl) that are not on your target database (URL).
2. Unexpected: these are objects on our target database (url) that are not on your source database (referenceUrl)
3. Changed: the object as it exists on the source database (refernceUrl) is different than as it exists in the target database (url)

Liquibase Community diff categories include:

1. Catalog
2. Column
3. Foreign Key
4. Index
5. Primary Key
6. Schema
7. Sequence
8. Procedure
9. Unique Constarints
10. View

Filtering diff types

Liquibase allows us to use diffType attribute to filter the types of objects we want to compare.

Multiple filters can be added to the attribute as a comma seperated list.

If no diffTypes are specified, all objects are considered.

liquibase --outputFile=db\_vs\_qa\_tables.txt --diffTypes=tables diff

diffChangeLog command

diffChnageLog command allows us to receive information on differences between two databases we are comparing an creates as changelog file containing deployable changesets.

The diffChangeLog command is typically used when you want to create a deployable *changelog* to synchronize multiple databases. The diffChangeLog command also provides more information about:

* Missing objects in your database
* Changes made to your database
* Unexpected items in your database

Running the diffChangeLog command

Running the diff command requires two URLs.

referenceURL- the source for comparison.

url- target of the comparison.

Sample of liquibase.properties file

url=jdbc:mysql://localhost:3306/qa?autoReconnect=true&amp;useSSL=false

username=root

password=root

classpath=D:/Sony/Learn/Liquibase/Connector/mysql-connector-java-5.1.48-bin.jar

referenceUrl=jdbc:mysql://localhost:3306/db?autoReconnect=true&amp;useSSL=false

referenceUsername=root

referencePassword=root

liquibase --changeLogFile=diff\_2.mysql.sql diffChangeLog

Database rollback commands

rollback <tag> command

Rolls back th database to the state it was in when the tag was applied.

Sample changelog test.sql

--liquibase formatted sql

--changeset t14:12

create table t14 (

    id int primary key,

    name varchar(255) ,

    test varchar(100)

);

-- rollback drop table t14;

--changeset t15:12

create table t15(

    id int primary key,

    name varchar(255) ,

    test varchar(100)

);

-- rollback drop table t15;

--changeset t16:12

create table t16(

    id int primary key,

    name varchar(255) ,

    test varchar(100)

);

-- rollback drop table t16;

--changeset t17:12

create table t17(

    id int primary key,

    name varchar(255) ,

    test varchar(100)

);

-- rollback drop table t17;

--changeset t18:12

create table t18(

    id int primary key,

    name varchar(255) ,

    test varchar(100)

);

-- rollback drop table t18;

To specify tag, we can use

liquibase tag <tag\_name>

Suppose, there are two tags, when we rollback to v1, all changes made after releasing tag v1 will get deleted and entry in DATABASECHANGELOG will also get deleted.

liquibase --changeLogFile=test.sql rollback v1

rollbackSQL <tag> command

It is a helper command that produces the raw SQL Liquibase that would run when using the rollback<tag> command.

It is best practice to use the rollbackSQL <tag> command before running the rollback <tag> command to ensure that we eliminate any potential risks.

liquibase --changeLogFile=test.sql –outputFile=file\_name.sql rollbackSQL v1

If outputFile attribute is not provided, result will be displayed in STDOUT.

rollbackToDate command

The rollbackTo Date command reverts database to the state it was in at the date and time you specify.

This command is used when we want to revert all changes made to our database from today’s date to the date and time we specify.

It reverts those changesets to their previous state and allows us to target the date and time without impacting changes or deployments that came before the date and time specified.

Run this command as

liquibase --changeLogFile=test.sql rollbackToDate 2021-03-08T13:21:51

liquibase --changeLogFile=test.sql rollbackToDate 2021-03-08 13:21:51

rollbackToDateSQL command

It is a helper command that allows us to inspect the SQL liquibase will run while using the rollbackToDate command.

The rollbackToDateSQL command is typically used when you want to inspect the raw SQL before running the rollbackToDate command, so you can correct any issues that may arise before running the command.

Run this command as

liquibase --changeLogFile=test.sql rollbackToDateSQL –outputFile=file\_name.sql 2021-03-08T13:21:51

liquibase --changeLogFile=test.sql rollbackToDateSQL –outputFile=file\_name.sql 2021-03-08 13:21:51

If outputFile attribute is not specified, rollbackToDate SQL output goes to console (STDOUT).

rollbackCount <value> command

This command is used to roll back to last <value> changesets.

It is used when we want to roll back changes sequentially, starting with most recent changes and working backward until the value specified is reached.

Run this command as

liquibase --changeLogFile=test.sql rollbackCount 2

rollbackCountSQL <value> command

It is a helper command that allows us to inspect the SQL Liquibase will run while using the rollbackCount <value> command.

It is used when we want to inspect the raw SQL before running the rollbackCount<value> command, so we can correct any issues that may arise before running the command.

Run this command as

liquibase --changeLogFile=test.sql –outputFile=file\_name.sql rollbackCountSQL 2

If outputFile attribute is not specified, rollbackToDate SQL output goes to console (STDOUT).

rollbackOneUpdate command

This command reverts (rolls back) all changesets related by a specific deploymentId that was made during a previous change to our database.

It is available for Liquibase Pro users only.

Run this command as:

liquibase rollbackOneUpdate --deploymentId=068379006

rollbackOneUpdateSQL command

It is a helper command that allows us to inspect SQL liquibase will run to revert all changeset associated with the deplymentID specified in the rollbackOneUpdate command.

It is available only for Liquibase Pro users.

Run this command as:

liquibase rollbackOneUpdateSQL –outputFile=file\_name.sql --deploymentId=068379006

If outputFile attribute is not specified, rollbackToDate SQL output goes to console (STDOUT).

futureRollbackSQL command

It is a helper command that produces the raw SQL liquibase would need to roll back changes that have not yet been deployed yet to our database.

Note:

UpdateSQL command is a haleper command that we can use before running the update command. The main difference is that updateSQL creates objects associated with undeployed changesets, and futureRollbackSQL drops objects associated with undeployed changesets.

Run this command as:

liquibase --changeLogFile=test.sql –outputFile=file\_name.sql futureRollBackSQL

If outputFile attribute is not specified, rollbackToDate SQL output goes to console (STDOUT).

updateTestingRollback command

This command tests rollback support by deploying all pending changesets to the database, executes a rollback seuqnetially for the equal number of changesets that were deployed, and then runs the update again deploying all changesets to the database.

updateTestingRollback is typically used when you want to test rollback functionality when deploying *changesets* in your *changelog* sequentially.

updateTestingRollback utilizes a multi-step operation and runs in sequential order:

1. update changeset1; update changeset2; update changeset3
2. rollback changeset3; rollback changeset2; rollback changeset1
3. update changeset1; update changeset2 update changeset3

Run this command as:

liquibase --changeLogFile=test.sql updateTestingRollback

Maintenance commands

changelogSync command

This command is used to mark all undeployed changes in our changelog as executed in our database.

Deploy nabhayeka changelogs lai database ma executed bhanera dekhaucha.

An example use case for the changelogSync command is when you have a DEV environment with a set of objects used only in DEV, and you want to use the same *changelog* to manage a new TEST environment.

The TEST environment does not have or need, those DEV-only objects. To avoid deploying the DEV-only objects, you run the changelogSync command to mark those changes as executed in the DATABASECHANGELOG which tells Liquibase to treat these databases as equivalent.

You can also use the changeLogSync command to mark a change as executed if the object associated with the change was created manually on the database. By marking the *changeset* as executed, it prevents the next Liquibase update from failing as it tries to create an object that already exists.

Run this command as:

liquibase --changeLogFile=test.sql changelogSync

changelogSyncSQL command

The changelogSyncSQL is a helper command that produces the raw SQL Liquibase would run when using the changelogSync command.

Run this command as:

liquibase --changeLogFile=test.sql –outputFile=file\_name.sql changelogSyncSQL

If outputFile attribute is not specified, rollbackToDate SQL output goes to console (STDOUT).

clearCheckSums command

This command is used to clear all checksums and nullifies the MD5SUM column of DATABASECHANGELOG table so they will be recomputed on the next database update.

Run this command as:

liquibase --changeLogFile=test.sql clearCheckSums

dropAll command

This command is used to drop all database objects owned by the user.

All tables including DATABASECHANGELOG and DATABASECHANGELOGLOCK will be deleted.

Run this command as:

liquibase --changeLogFile=test.sql dropAll

listLocks command

This comman returns hostname, IP address and timestamp the liquibase lock was added to the DATABASECHANGELOGLOACK table. If no lock is available, a message stating No Lock is displayed.

Run this command as:

liquibase --changeLogFile=test.sql listLocks

releaseLocks command

This command is use to remove specific Liquibase record from the DATABASECHANGELOGLOCK table in the needed database.

Run this command as:

liquibase --changeLogFile=test.sql releaseLocks

status –verbose command

This command produces a list of pending changesets with additional information that includes id, author and file path name.

It does not modify the database.

The status --verbose command is typically used when *changesets* were added to a *changelog* through source control by another developer. The command confirms what has been deployed and what *changesets* are pending per author and corresponding ids.

Run this command as:

liquibase --changeLogFile=test.sql status –verbose

snapshot command

It captures the current state of URL database, which is the target database.

The snapshot command has two modes:

* When run without options, it gathers the current state of the database and shows a text-based version of the schema to STDOUT.
* When run with the --snapshotFormat=JSON option, it creates a JSON file that represents the current state of the URL database. Alternatively, you can have a YAML-based output by using the --snapshotFormat=yaml attribute.

Run this command as:

liquibase snapshot

liquibase --outputFile=snap.json snapshot --snapshotFormat=json

liquibase --outputFile=snap.yaml snapshot --snapshotFormat=yaml

tag <tagstring> command

This command is used to mark the current database state so that we can roll back changes in future.

Run this command as:

liquibase tag v1

tagExists <tag string> command

This command is used to check whether the tag specified already exists in our database or not.

Run this command as:

liquibase tagExists v1

validate command

This command is used to check and identify any possible errors in a changelog that can cause the update command to faile.

The validate command is mainly used when you want to detect if there are any issues with a *changelog* before running the update command.

Use the validate command to ensure that:

* The XML/YAML/JSON/formatted SQL is structured correctly
* Referenced files can be found
* There are no duplicated id/author/file combinations
* There aren't any checksum errors
* Any required or not allowed attributes are correct for your database

Run this command as:

liquibase --changeLogFile=test.sql validate

markNextChangeSetRan command

The markNextChangeSetRan command marks the next change you apply as executed in your database.

The markNextChangeSetRan command is typically used when you have made a change manually, and the deployment is failing. Here is a use case that shows the whole process in more detail:

* You have a *changelog* with a specific *changeset* and want to create a table in your database, and then apply your changes manually without using Liquibase. In this case, there will be no record of this change in the DATABASECHANGELOG table.
* Then you decide to deploy the same *changeset* by using the update command, Liquibase checks the DATABASECHANGELOG table to see if there is such a *changeset*. Since there is no record of it, Liquibase tries to create a table, but as it already exists, you receive an error.
* As a result, Liquibase stops deployment at that specific *changeset* without executing it.
* Running the markNextChangeSetRan adds that *changeset* in the *changelog* as a record even though the table is already created. The command detects that the *changeset* was deployed.

Run this command as:

liquibase --changeLogFile=test.sql markNextChangeSetRan

Documentation commands

This commmand is used to generate documentation in a Javadoc format based on the existing database and changelogs.

Run this command as:

liquibase --changeLogFile=test.sql directory\_name changelogDocs

where,

directory\_name is the directory where the database documentation is generated.

Liquibase Community Workflows

Creating and confihuring a liquibase.properties file

Liquibase.properties file is a text-based file that allows us to store properties that do not change often.

We can typiclly use this file to specify our database connection information.

This saves our time and potential typing errors by avoiding need to enter those properties as command line arguments.

| **Property** | **Definition** |
| --- | --- |
| changeLogFile | Specifies the path to the *changelog* to execute. |
| driver | Specifies the driver class name for your target database. |
| referenceUrl | Specifies the source database for performing comparisons. |
| username | Specifies the username for your target database. |
| password | Specifies the password for your target database. |
| referenceDriver | Specifies the driver class name for your source database. |
| url | Specifies the database you want to use to compare to your source database. Also known as your target. |
| referenceUsername | Specifies the username for your source database. |
| referencePassword | Specifies the password for your source database. |
| liquibaseProLicenseKey | Specifies your Liquibase Pro license key (If you have one). |
| classpath | Specifies the directories and jar files to search for *changelog* files and custom extension classes. Multiple directories can be separated with ; on Windows or : on Linux or MacOS. |

Defining Multiple Schemas in Liquibase

Liquibase allows handling multiple schemas with the following commands:

1. Diff command
2. diffChangeLog command
3. generateChangeLog command
4. snapshot command

snapshot command

While running snapshot comand on multiple schemas, enter the –schemas flag after the snapshot command.

Run this command as:

liquibase --outputFile=snap\_3.json snapshot --snapshotFormat=json --schemas=qa,test