Refactoring von Datenbankschemas mit Liquibase

Fließender Übergang

Software verändert sich, und mit ihr verändern sich die benötigten Datenstrukturen. Komplexe SQL-Skripte sind meist die Folge. Alternativ dazu existiert ein einfach zu erlernendes, aber mächtiges Werkzeug. **Dirk Mahler**

Auf einen Blick

Inhalt
Mit Liquibase werden auf einfache Weise Änderungen im Domänenmodell auf Schemas relationaler Datenbanken übertragen, ohne bestehende Strukturen und bereits vorhandene Daten zu zerstören.

Plattform
Datenbanken allgemein, Java

Voraussetzungen
Java 5, Liquibase, Datenbank mit JDBC-Treiber (im Beispiel H2), Windows oder Unix

Autor
Dirk Mahler ist Senior Consultant der buschmais GbR in Dresden.

Nichts ist so beständig wie der Wandel". Dieses oft benutzte Zitat trifft insbesondere auch auf den Lebenszyklus von Software zu. Es ist nahezu unmöglich, eine Anwendung vollständig auf dem Reißbrett zu entwerfen, sie exakt in der geplanten Form umzusetzen und anschließend nicht mehr verändern zu müssen. Neue ebenso wie sich verändernde Anforderungen erzwingen zu verschiedenen Zeitpunkten im Entwicklungsprozess entweder Ergänzungen oder Änderungen, denen praktisch jeder Teil des Systems unterliegt. Der Begriff Refactoring hat mittlerweile einen festen Platz im Vokabular der Softwareentwickler gefunden und wird durch Entwicklungsumgebungen für gängige Programmiersprachen sehr gut unterstützt.

Bestimmung auszuführender ChangeSets (Bild 1)

Refactoring, Domänenmodell und Schemas

Während diese Aussage ohne Einschränkung auf die Programmiersprache Java und ihre Werkzeuge (zum Beispiel Eclipse) angewendet werden kann, klafft ausgerechnet in deren näherem Umfeld eine Lücke: Wie werden Änderungen im Domänenmodell auf Schemas relationaler Datenbanken übertragen, ohne bestehende Strukturen und bereits vorhandene Daten zu zerstören? Erfahrungsgemäß existieren unter anderem die folgenden Szenarien:

- Während des Entwicklungsprozesses muss ein Entwickler auf einen anderen Stand der Software (zum Beispiel Branch oder Tag) wechseln.
- Ein neues Release einer Anwendung soll ausgeliefert werden, eine Migration der Datenbank des Kunden ist notwendig. Neben Migrationsskripten soll eine Dokumentation der anstehenden vorgenommenen Änderungen ausgeliefert werden.

Ein gängiges Vorgehen ist die Pflege von Skriptstrukturen, welche die notwendigen strukturellen Anpassungen sowie Änderungen an Datenbeständen in Form von SQL-Statements beinhalten. Bei der Realisierung müssen in der Regel die folgenden Probleme gelöst werden:

- Wie werden die Migrationsskripte organisiert, um Synchronität des Datenbankschemas mit dem Versionsstand der Software zu garantieren?
- Gibt es Abhängigkeiten zwischen einzelnen Skripten, das heißt, in welcher Reihenfolge müssen sie ausgeführt werden?
- Auf welchem Stand befindet sich ein existierendes Schema, das heißt, welche Migrationsschritte wurden in einer konkreten Umgebung bereits ausgeführt?
- Welches Risiko stellen im Entwicklungsprozess parallel auftretende Änderungen dar?
- Wie können Änderungen effizient auf verschiedene Datenbanksysteme abgebildet werden (zum Beispiel Oracle als Produktionsdatenbank und H2 als lokale Entwicklungsdatenbank)?

Mit Liquibase (*www.liquibase.org*) existiert eine Open-Source-Lösung unter LGPL-Lizenz, welche ein Vorgehen sowie technische Unterstützung für die Migration von Datenbankschemas anbietet. Sie richtet sich zwar primär an Entwickler von Java-Anwendungen, kann jedoch auch in anderen Umfeldern problemlos eingesetzt werden.

Listing 1: Das erzeugte XML-Dokument

```
<databaseChangeLog …>
  <changeSet author="dirk.mahler (generated)" id="1233124414681-1">
    <createTable tableName="PERSON">
      <column name="ID" type="INTEGER">
        <constraints nullable="false" primaryKey="true"
          primaryKeyName="CONSTRAINT_8"/>
      </column>
      <column name="FIRSTNAME" type="VARCHAR(50)"/>
      <column name="LASTNAME" type="VARCHAR(50)"/>
    </createTable>
  </changeSet>
</databaseChangeLog>
```

Das Konzept: DatabaseChangeLogs

Die grundlegende Idee von Liquibase liegt in der Verarbeitung sogenannter DatabaseChangeLogs, welche als XML-Dateien vorliegen und jeweils eine Sequenz ChangeSets beziehungsweise Verweise auf weitere DatabaseChangeLogs als Include-Anweisungen beinhalten (Listing 2).

Jedes ChangeSet ist eindeutig identifizierbar und umfasst eines oder mehrere vorzunehmende Refactorings, zum Beispiel das Anlegen, Ändern und Löschen von Tabellen sowie die Manipulation von Daten.

Innerhalb des zu migrierenden Datenbankschemas wird von Liquibase selbst eine Tabelle namens *DATABASECHANGELOG* verwaltet, welche Meta-Informationen über die bereits angewandten ChangeSets beinhaltet. Somit muss zur Bestimmung ausstehender Änderungen nur die Differenzmenge zum vorgegebenen DatabaseChangeLog in der gerade vorliegenden XML-Datei ermittelt werden.

Die Refactorings selbst sind weitestgehend unabhängig von einer konkreten Datenbank beschrieben. Sie werden von Liquibase in SQL-Statements übersetzt und je nach Wunsch entweder unmittelbar auf dem Schema ausgeführt (Entwicklung) oder nur die entsprechenden Skripte dafür erzeugt (Auslieferung). Dabei wird eine breite Palette gängiger kommerzieller und quelloffener Datenbanken (zum Beispiel Oracle, DB2, MySQL, H2 etc.) unterstützt.

Eine wichtige Voraussetzung für die Funktionsfähigkeit des beschriebenen Ansatzes ist die Bestimmung der Identität eines ChangeSets. Dazu werden folgende Informationen miteinander kombiniert (Bild 1):

- der Dateiname des XML-DatabaseChangeLogs, welches das Changeset beinhaltet,
- der Name des Autors,
- eine vom Autor vergebene Id des ChangeSets.

Der Migrationsprozess kann sowohl manuell auf der Kommandozeile als auch automatisch innerhalb eines Build-Prozesses (Ant-Task, Maven-Plug-in) oder beim Start einer Anwendung (zum Beispiel ServletContextListener) durchgeführt werden.

Listing 2: Aufbau eines ChangeLogs

```
<?xml version="1.0" encoding="UTF-8"?>
<databaseChangeLog xmlns="http://www.liquibase.org/xml/ns/
dbchangelog/1.9" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance" xsi:schemaLocation="http://www.liquibase.org/xml/
ns/dbchangelog/1.9 http://www.liquibase.org/xml/ns/dbchangelog/
dbchangelog-1.9.xsd">

  <changeSet id="1" author="dirk.mahler">
    <!–refactorings –>
    <createTable tableName="person">
      <column name="id" type="int">
        <constraints primaryKey="true" nullable="false"/>
        </column>
        <column name="lastname" type="varchar(50)">
          ...
    </createTable>
  </changeSet>

  <changeSet id="2" author="dirk.mahler">
    ...
  </changeSet>

  <include file="branding/changelog.xml" />
</databaseChangeLog>
```

Arbeiten mit Netz und doppeltem Boden

Zur Vermeidung von Inkonsistenzen kommen verschiedene Sicherungsmechanismen zum Einsatz. Beim Einlesen der XML-Beschreibungen wird über jedes enthaltene ChangeSet eine MD5-Checksumme gebildet und mit den in der Datenbank hinterlegten Meta-Informationen verglichen. So können nachträgliche Änderungen an einem ChangeSet in der XML-Datei erkannt werden und Liquibase ist in der Lage, mit einer Fehlermeldung oder erneuter Ausführung zu reagieren. Das gewünschte Verhalten kann für jedes einzelne ChangeSet definiert werden.

Zur Sicherstellung einer Alles-oder-nichts-Semantik werden sämtliche Refactorings eines ChangeSets innerhalb einer Transaktion ausgeführt, um bei Fehlern während der Migration zum Ausgangszustand zurückzukehren. Dem sind allerdings technische Grenzen seitens ▶

Listing 3: Ein weiteres ChangeSet

```
<databaseChangeLog …>
  <changeSet author= dirk.mahler (generated)"…
  </changeSet>

  <changeSet author="dirk.mahler" id="1">
    <addColumn tableName="person">
      <column name="age" type="int"/>
    </addColumn>
  </changeSet>
</databaseChangeLog>
```

der Datenbanken gesetzt. Das Erzeugen, Ändern beziehungsweise Löschen von Tabellenstrukturen wird von ihnen gegebenenfalls außerhalb eines solchen Kontexts ausgeführt und kann daher im Zweifelsfall nicht zurückgerollt werden.

Um konkurrierende Änderungen verschiedener gleichzeitig aktiver Liquibase-Instanzen auszuschließen wird eine weitere Tabelle mit Meta-Informationen namens *DATABASECHANGELOGLOCK* verwaltet, in der Sperrinformationen für den Zeitraum einer laufenden Migration hinterlegt werden.

Schemamigration am praktischen Beispiel

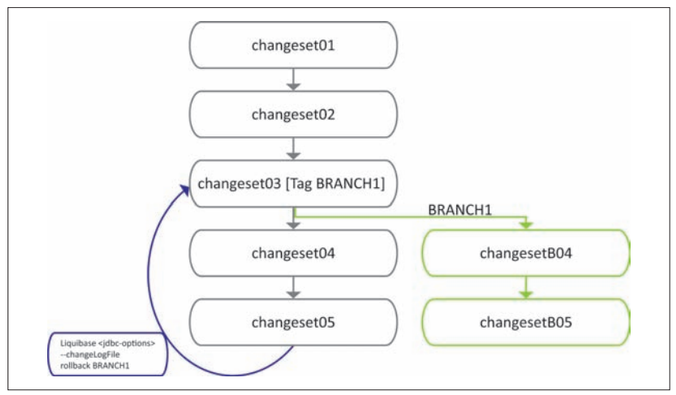
Im Folgenden wird zur Veranschaulichung der Arbeit mit Liquibase ein Szenario bestehend aus verschiedenen Migrationsschritten beschrieben. Zum Einsatz kommt die Kommandozeilenversion des Werkzeugs, welches nach dem Herunterladen und Entpacken der auf der Webseite bereitgestellten ZIP-Datei sofort zur Verfügung steht. Darüber hinaus wird der JDBC-Treiber der zur migrierenden Datenbank benötigt. Im Beispiel wird mit einer H2-Instanz gearbeitet *(www.h2database.com)*. Für diese lässt sich nach deren wirklich sehr einfachen Installation eine Anwendung namens H2-Console starten. Sie erlaubt eine Überwachung der vorgenommenen Änderungen.

Die für den Verbindungsaufbau notwendigen JDBC-Parameter müssen als Kommandozeilenoptionen an Liquibase übergeben werden:

```
liquibase –classpath=▷
h2-1.1.107.jar –driver=org.h2.Driver ▷
–url="jdbc:h2:tcp://localhost/▷
~liquibase" –username=sa –password=sa▷
 –changeLogFile=<databaseChangeLog>▷
 <command>
```

Auf die immer wiederkehrenden JDBC-Angaben wird in den folgenden Beispielen verzichtet, sie werden durch *<jdbc-options>* ersetzt. Alternativ besteht die Möglichkeit der Verwendung einer Properties-Datei, in der die Daten hinterlegt sind. Bei der Integration von Liquibase in Build-Systeme oder Anwendungen werden die jeweils vorhandenen Mechanismen zur Konfiguration unterstützt.

SCM-Unterstützung
(Bild 2)



Listing 4: Hinzufügen von tagDatabase

```
<changeSet author="dirk.mahler" id="2">
  <tagDatabase tag="BRANCH1"/>
</changeSet>
```

Aller Anfang ist leicht

Entsteht ein Projekt auf der viel beschworenen grünen Wiese, so wird naturgemäß mit einem leeren DatabaseChangeLog begonnen, in das zunächst ChangeSets mit Refactorings zum Erzeugen von Tabellen aufgenommen werden.

Viel interessanter ist jedoch die Frage, wie mit bereits bestehenden Informationen eines Schemas umgegangen werden soll? Für das Beispiel liegt in der H2-Instanz bereits eine Tabelle vor, die bereits per H2-Console erzeugt wurde:

```
create table person(id int, firstname
varchar(50), lastname varchar(50),
primary key (id));
```

Für die Generierung eines initialen DatabaseChangeLogs bietet Liquibase das Kommando *generateChangeLog* an:

```
liquibase <jdbc-options> ▷
–changeLogFile=changelog.xml ▷
generateChangeLog
```

Das erzeugte XML-Dokument *changelog.xml* beinhaltet bereits ein ChangeSet und ist in Listing 1 abgebildet.

Es folgt die Probe aufs Exempel, indem zunächst die Struktur der Datenbank verworfen wird:

```
drop table person;
```

Die Informationen sollen durch Liquibase neu aufgebaut werden. Anders ausgedrückt, das Schema soll auf den Stand aktualisiert werden, der dem DatabaseChangeLog in der XML-Datei entspricht. Dafür wird das Kommando *update* ausgeführt:

```
liquibase <jdbc-options> ▷
–changeLogFile=changelog.xml update
```

Ein Blick in die H2-Console offenbart, dass neben der gewünschten Tabelle *PERSON* die zwei von Liquibase verwalteten Tabellen erzeugt wurden. *DATABASECHANGELOG* enthält darüber hinaus eine Zeile, die das angewandte ChangeSet beschreibt.

Während der weiteren Entwicklung der Anwendung soll der Java-Klasse, welche die Tabelle *PERSON* repräsentiert, ein Attribut *age* hinzugefügt werden. Diese Änderung muss sich auch in der Datenbank widerspiegeln und erfordert das Hinzufügen eines weiteren ChangeSets (Listing 3).

Der Aktualisierungsprozess wird wiederum per *update* ausgelöst. Die Tabelle *person* wird dabei um die gewünschte Spalte ergänzt und die Tabelle *DATABASECHANGELOG* mit der Information über das angewandte ChangeSet aktualisiert.

Rolle rückwärts

In bestimmten Fällen ist es notwendig, das Datenbankschema auf einen älteren Stand zurückzusetzen. Für viele der von Liquibase angebotenen Refactorings wird dieses Ansinnen direkt unterstützt.

Im zuletzt eingeführten ChangeSet wurde der Tabelle *PERSON* eine Spalte *age* hinzugefügt; das soll rückgängig gemacht werden. Das Kommando *rollbackCount* erlaubt das Zurückrollen um eine vorgegebene Anzahl von Schritten. Hier entspricht das exakt einem ChangeSet:

```
liquibase <jdbc-options> ▷
–changeLogFile=changelog.xml ▷
rollbackCount 1
```

Eine Überprüfung der Struktur ergibt, dass die Spalte *age* und der entsprechende Eintrag in der Tabelle *DATABASECHANGELOG* entfernt wurden. Für das Zurückrollen werden zwei weitere Kommandos unterstützt:

- *rollbackToDate <date/time>*: Rollt auf den Stand eines angegebenen Zeitpunkts zurück und erwartet dafür einen Parameter in der Form yyyy-MM-dd HH:mm:ss.
- *rollback <tag>*: Setzt das Schema auf einen Stand zurück, der zuvor mit einem Tag versehen wurde.

Das erste Szenario bedarf keiner näheren Erläuterung. Ein Blick auf die zweite Variante ist allerdings interessant, da sie den Umgang mit Versionskontrollsystemen (SCM) wie CVS und Subversion (SVN) auf Datenbanken abbildet.

Am Scheideweg

Die Ausführung von *update* bringt zunächst das Schema wieder auf den neuesten Stand, das heißt, *PERSON* erhält wieder die Spalte *age*. Nehmen wir an, dass die XML-Datei des DatabaseChangeLogs im Beispiel durch SVN verwaltet wird und sich derzeit im Trunk befindet. Im Laufe des Entwicklungsprozesses wird beschlossen, einen Zweig namens *BRANCH1* zu eröffnen. Das geschieht zunächst im Versionskontrollsystem selbst. Im DatabaseChangeLog wird dafür ein entsprechendes ChangeSet namens *tagDatabase* hinzugefügt (Listing 4) und per *update* auf das Schema angewandt.

Listing 5: Datensatz in Tabelle PERSON

```
<changeSet author="dirk.mahler" id="3">
  <insert tableName="person">
    <column name="id" valueNumeric="1"/>
    <column name="firstname" value="Max"/>
    <column name="firstname" value="Mustermann"/>
  </insert>
  <rollback>
    <delete tableName="person">
      <where>id=1</where>
    </delete>
  </rollback>
</changeSet>
```

Die einzige sichtbare Änderung ist ein neuer Eintrag in der Tabelle *DATABASECHANGELOG* sowie die Markierung des zuletzt angewandten ChangeSets (Hinzufügen der Spalte *age*) mit dem Tag *BRANCH1*.

Die Arbeit an der Beispielanwendung soll zunächst weiterhin im Trunk erfolgen. In dessen XML-Datei wird ein ChangeSet eingetragen, das einen Datensatz in die Tabelle *PERSON* einfügt (Listing 5). Dieser Schritt ist nur für diesen Entwicklungszweig relevant und kann mit *update* angewandt werden.

Tritt der Fall ein, dass der Entwicklungszweig gewechselt werden muss, kann das bereits genannte Kommando *rollback <tag>* ausgeführt werden (Bild 2):

```
Liquibase <jdbc-options> ▷
–changeLogFile=changelog.xml ▷
rollback BRANCH1
```

Die Datenbank befindet sich damit auf dem Stand des Trunks vor der Erzeugung des Zweiges. Jetzt kann der Entwickler SVN anweisen und seine Dateien mit dem Stand aus *BRANCH1* aktualisieren. Dabei werden die in diesem Zweig vorliegenden Änderungen am XML-DatabaseChangeLog abgeholt und können anschließend über das *update*-Kommando auf das Schema übertragen werden.

Das letzte beschriebene ChangeSet beinhaltet ein Tag namens *<rollback>*. Dieses beschreibt die für ein Zurückrollen des ChangeSets notwendigen Änderungen an der Datenbank. Eine An- ▶

Listing 6: Weiteres ChangeSet

```
<changeSet author="dirk.mahler" id="4">
  <sql>update person set id=id+1</sql>
  <rollback>
    <sql>update person set id=id-1</sql>
</rollback>
</changeSet>
```

gabe dieses Tags ist insbesondere dann nötig, wenn Liquibase keine explizite Rollback-Unterstützung für ein Refactoring anbietet oder anbieten kann. Letzteres ist zum Beispiel der Fall für *insertData*.

Automatisierung oder Handarbeit

Im vorliegenden Beispiel wurden bereits einige der von Liquibase unterstützten Refactorings demonstriert. Sie stellen aber nur einen kleinen Bruchteil der verfügbaren Funktionalitäten dar. Diese werden in Kategorien aufgeteilt, die in Tabelle 1 beschrieben sind.

Die Kategorie *Custom Refactorings* ist von Interesse, wenn kein vordefiniertes Refactoring zur Verfügung steht. Als Ausweg steht die Aufnahme von SQL-Statements in ein ChangeSet offen.

In unserem Beispiel sollen dazu die Werte der Spalte *Id* aus der Tabelle *PERSON* inkrementiert werden. Dafür wird das ChangeSet in Listing 6 erzeugt.

Achtung! Der Einsatz dieser Möglichkeit geht unter Umständen zulasten der Portabilität des DatabaseChangeLogs, also kann es zum Beispiel nur noch auf Oracle-Datenbanken ausgeführt werden.

Immer? Nicht immer ...

Bisher wurde davon ausgegangen, dass jedes ChangeSet immer ausgeführt werden kann. In manchen Fällen ist es jedoch durchaus sinnvoll, einen Teil der Änderungen nur in einem bestimmten Kontext oder unter bestimmten Bedingungen auszuführen. Dazu können ChangeSets mit entsprechenden Informationen angereichert werden:

```
<changeSet author="dirk.mahler" id="3"
context="test">
```

Tabelle 1: Refactoring-Kategorien

Kategorie	Beschreibung
Structural Refactorings	Anlegen, Bearbeiten und Löschen von Tabellen, Views und Stored Procedures
Data Quality Refactorings	Verwalten von Constraints, Sequenzen, Auto-Inkrement-Spalten und Default-Values
Referential Integrity Refactorings	Hinzufügen und Entfernen von Constraints für Primär- und Fremdschlüssel
Non-Refactoring Transformations	Einfügen, Bearbeiten und Entfernen von Daten sowie Tagging
Architectural Refactorings	Verwalten von Indizes
Custom Refactorings	SQL-nahe Operationen, die entweder die von Liquibase erzeugten Statements manipulieren beziehungsweise komplett durch den Nutzer definiert sind.

Listing 7: Tag <preConditions>

```
<preConditions onFail="WARN">
  <dbms type="oracle" />
  <runningAs username="SYSTEM" />
</preConditions>
```

Die Ausführung des Kommandos *update* kann dazu mit einer Option erfolgen, die den jeweiligen Kontext definiert:

```
liquibase <jdbc-options> ▷
–changeLogFile=changelog.xml ▷
-contexts=test
```

Weitere Bedingungen können auf der Ebene von ChangeSets beziehungsweise kompletten DatabaseChangeLogs mithilfe des Tags *<preConditions>* formuliert werden (Listing 7).

In der Praxis

Genau genommen löst Liquibase keine Probleme, die nicht auch durch eine geschickte Organisation von SQL- und Build-Skripten gelöst werden könnten.

Der Einsatz dieses Werkzeuges erlaubt aber bei überschaubarem Installations- und Konfigurationsaufwand sowie geringem Lernbedarf die Anwendung verschiedener sinnvoller und in der Praxis erprobter Szenarien: Zurückrollen von Änderungen, Unterstützung der Arbeit mit Versionskontrollsystemen und weitestgehende Datenbankunabhängigkeit. Dazu kommen weitere Möglichkeiten zur Generierung von Diff-Skripten sowie Reports über den aktuellen Stand des Schemas beziehungsweise noch ausstehender Migrationsschritte.

Einige Details stellen sich als problematisch heraus: Die beschriebene Identifizierung eines ChangeSets unter Zuhilfenahme des Dateinamens ist in modularisierten Projekten (zum Beispiel unter Nutzung von Maven) eventuell problematisch. Die Verwendung eines Namespace-orientierten Ansatzes erscheint hier wünschenswert. Ein weiteres Problem ist das zwingende Vorhandensein von Schreibrechten auf der Datenbank, selbst dann, wenn auszuführende Änderungen nur als Skripte erzeugt werden sollen (so zum Beispiel zur Vorbereitung der Migration einer Produktionsdatenbank). Dafür leistet die Dokumentation von Liquibase sehr gute Unterstützung bei der Arbeit, die – anders als von vielen Open-Source-Lösungen gewohnt – wirklich verständlich und brauchbar ist. **[ef]**

[1] Homepage des Liquibase-Projekts; www.liquibase.org
[2] Homepage der H2-Datenbank; www.h2database.com