Continuous Database Integration mit Flyway

Sandra Parsick, Freiberufler

Skripte für relationale Datenbanken werden von Entwicklern gerne stiefmütterlich behandelt. Beim ersten Release können sie dank ORM-Frameworks generiert werden. Doch spätestens beim zweiten Release müssen Datenbank-Migrationsskripte geschrieben werden. Sie werden dann gerne an Tickets angehängt, per E-Mail verteilt oder in Release Notes versteckt. Irgendwann gibt es keinen Überblick mehr darüber, welche Datenbank-Skripte zu welcher Softwareversion gehören. Der Artikel zeigt, warum deren Einbindung in den Continuous-Integration-Prozess erstrebenswert ist, welche Voraussetzungen dafür geschaffen werden müssen und wie Flyway dabei helfen kann.

Wer kennt es nicht: Entwickler müssen sich eine Test-Datenbank teilen und kommen sich ständig in die Quere, da man Phantomfehlern hinterherjagt. Doch der eigentliche Grund war, dass sie sich gegenseitig die Datenbank-Struktur zerschossen haben. Fehler in der Produktion lassen sich nicht nachbilden, da sich die Test-Datenbank gravierend von der Datenbank in der Produktion unterscheidet. Dadurch haben auch Testläufe der Migrationsskripte keine Aussagekraft.

Werden verschiedene Test-Datenbanken gepflegt, dann weiß keiner so genau, welche Skripte gegen welche Datenbank liefen. Zu guter Letzt werden die Datenbank-Skripte in unterschiedlichen Systemen abgelegt und es ist nicht klar, in welcher Reihenfolge die Skripte ausgeführt werden müssen oder ob alle relevanten Skripte für den nächsten Livegang vorhanden sind. Diese Problematik möchte „Continuous Database Integration" angehen, indem ein Prozess geschaffen wird, bei dem zu jeder Zeit, wenn eine Änderung an den Datenbank-Skripten gemacht wurde, eine Datenbank mit ihren Testdaten erneuert werden kann.

Grundregeln für eine Continuous Database Integration

Um einen „Continuous Database Integration"-Prozess erfolgreich zu etablieren, sollten folgende Grundregeln eingehalten werden:

- Behandle den Datenbank-Code wie ganz normalen Source-Code
- Jeder Entwickler hat seine eigene Datenbank
- Die Test-Datenbanken ähneln den Produktions-Datenbanken
- Jede Änderung an der Datenbank ist nachvollziehbar

Um diese vier Grundregeln einhalten zu können, müssen folgende Maßnahmen ergriffen werden:

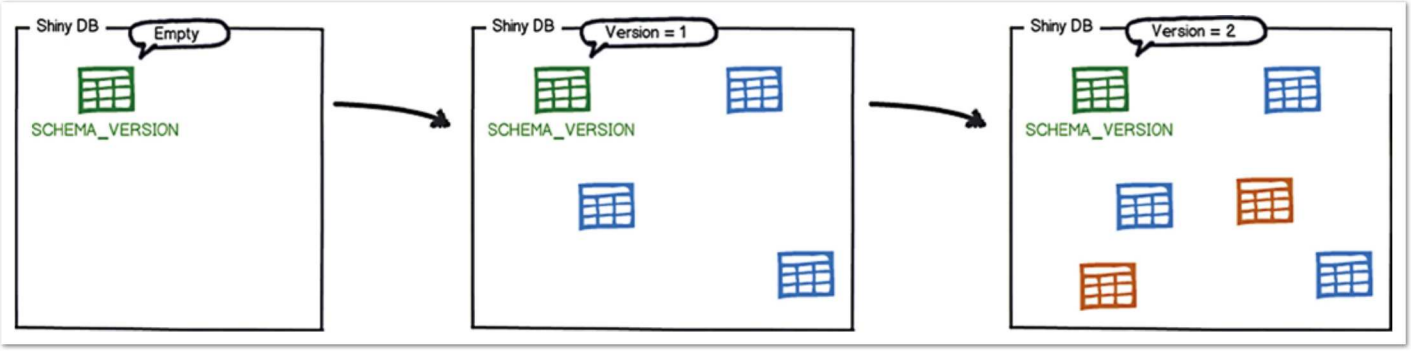

Abbildung 1: Arbeitsweise von Flyway bei einer leeren Datenbank (Referenz: „http://flyway.org")

schema_version

installed_rank	version	description	type	script	checksum	installed_by	installed_on	execution_time	success
1	1	Initial Setup	SQL	V1__Initial_Setup.sql	1996767037	axel	2016-02-04 22:23:00.0	546	true
2	2	First Changes	SQL	V2__First_Changes.sql	1279644856	axel	2016-02-06 09:18:00.0	127	true
3	2.1	Refactoring	JDBC	V2_1__Refactoring		axel	2016-02-10 17:45:05.4	251	true

Abbildung 2: Die Historien-Tabelle SCHEMA_VERSION (Referenz: „http://flyway.org")

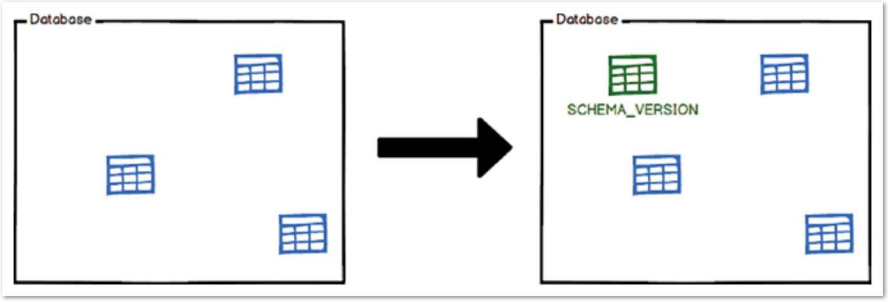

Abbildung 3: Flyway bei einer bestehenden Datenbank (Referenz: „http://flyway.org")

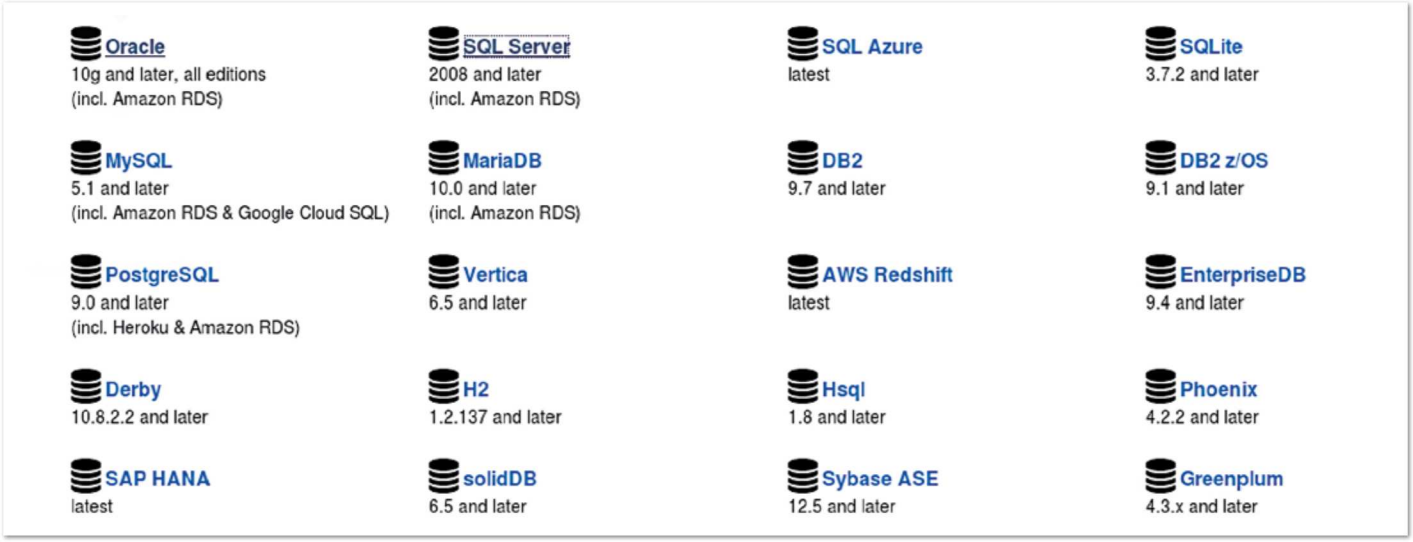

Abbildung 5: Unterstützte Datenbanken (Referenz: „http://flywaydb.org")

- Alle Datenbank-Artefakte wie DDL-Skripte, DML-Skripte, Konfigurationen, Testdaten, Skripte mit Stored Procedure oder Functions etc. gehören in ein Version-Control-System wie Subversion oder Git (Grundregel 1)
- Jede Änderung an den Datenbank-Artefakten wird getestet (Grundregel 1)
- Die Datenbank muss automatisiert aufsetzbar sein (Grundregel 2 und 3)
- Jede Änderung an der Datenbank wird in einer Historie protokolliert (Grundregel 4)

Diese Maßnahmen lassen sich gut mit Werkzeugen wie Liquibase, MyBatis oder Flyway umsetzen. Dieser Artikel zeigt, wie die Umsetzung mit Flyway aussehen könnte.

Flyway

Flyway *[1]* ist ein Migrations-Framework für relationale Datenbanken, geschrieben in Java. Das Ziel von Flyway ist, eine Datenbank „from scratch" erstellen zu können und den Stand der Datenbank zu verwalten. Dabei werden vier Migrationsmodi unterstützt: SQL- und Java-basierte sowie versionierte und wiederholbare Migrationen.

Die Grundfunktion von Flyway besteht darin, Migrationsskripte auszuführen („flyway migrate") und nach jedem erfolgreich ausgeführten Migrationsskript die Änderung in einer Tabelle in der betroffenen Datenbank zu protokollieren. Bei einer leeren Datenbank wird diese Historien-Tabelle („SCHEMA_VERSION") automatisch von Flyway angelegt; danach führt es die Migrationsskripte aus *(siehe Abbildung 1)*. In die Historien-Tabelle werden Informationen gespeichert *(siehe Abbildung 2)*:

- In welcher Reihenfolge die Skripte ausgeführt wurden („installed_rank")
- Welche Version die Skripte hatten („version")
- Eine Beschreibung dessen, was die Skripte machen („description")
- Um welche Art von Migration es sich handelt („type")
- Welches Skript ausgeführt wurde („script")
- Die Checksumme des Skripts, um nachträgliche Änderungen an den schon ausgeführten Skripten zu erkennen (nur bei SQL-basierten Skripten) („checksum")
- Von welchem Datenbank-Benutzer das Skript ausgeführt wurde („installed_by")
- Wann das Skript ausgeführt wurde („installed_on")
- Wie lange die Ausführung des Skripts dauerte („execution_time")
- Ob die Ausführung erfolgreich war („sucess")

	Versioniert	Wiederholbar
SQL-basiert	✔	✔
Java-basiert	✔	✔

Abbildung 4: Vier Möglichkeiten, um Migrationsskripte zu schreiben

Soll Flyway gegen eine schon bestehende Datenbank laufen, muss diese mit Flyway („flyway baseline") vorbereitet sein. Dabei wird die Historien-Tabelle „SCHEMA_VERSION" angelegt *(siehe Abbildung 3)*, danach kann die oben beschriebene Migration ausgeführt werden.

Vier Arten von Migrationsskripten

Wie am Anfang erwähnt, bietet Flyway vier Arten von Migrationsskripten an. Dabei sind immer zwei Arten miteinander kombiniert. *(siehe Abbildung 4)*.

Die versionierten Migrationsskripte haben immer eine eindeutige Version und werden im gesamten Lebenszyklus nur einmal ausgeführt. Typische Anwendungsfälle für diese Art von Skripten sind DDL-Änderungen (CREATE/ALTER/DROP für TABLES, INDEXES, FOREIGN KEYS etc.) oder einfache Datenänderungen.

Die wiederholbaren Migrationsskripte haben keine Version und werden immer dann ausgeführt, wenn sich ihre Checksumme ändert. Ihr Ausführungszeitpunkt ist, nachdem alle versionierten Skripte ausgeführt wurden. Typische Anwendungsfälle sind zum Beispiel die (Wieder-) Erstellung von Views, Procedures, Functions, Packages etc. oder ein Massen-Reimport von Stammdaten. Die SQL-basierten Migrationsskripte beinhalten SQL-Statements in einer Datenbank-spezifischen Syntax. Zusätzlich werden Platzhalter und Kommentare unterstützt *(siehe Listing 1)*.

Typische Anwendungsfälle für die SQL-Skripte sind DDL-Änderungen und einfache Datenänderungen. Die SQL-Syntax der Statements ist abhängig vom Datenbank-Anbieter, der eingesetzt wird. Momentan sind zwanzig Datenbank-Anbieter unterstützt *(siehe Abbildung 5)*. Der Dateiname der Skripte muss einem bestimmten Schema folgen, abhängig davon, ob die Skripte für eine versionierte oder eine wiederholbare Migration gedacht sind *(siehe Abbildung 6)*. Aus diesem Namensschema werden die Informationen für die Historien-Tabelle extrahiert („version", „description", „type" und „script").

```
/* Create a table for person */
Create table person (
  first_name varchar(128),
  last_name varchar(128)
);
GRANT SELECT, INSERT ON usermgnt.* TO 'technical-user'
@ '${address}' By '${password}';
```

Listing 1

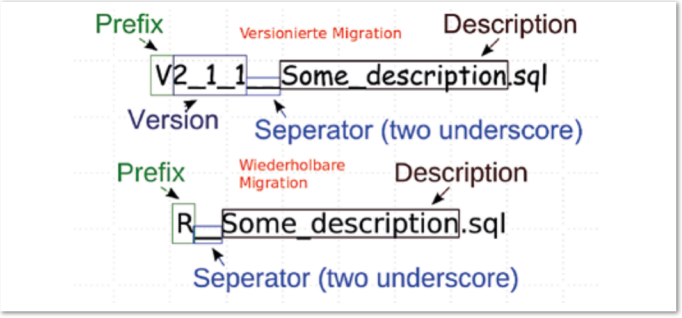

Abbildung 6: Bezeichnung der SQL-Skripte

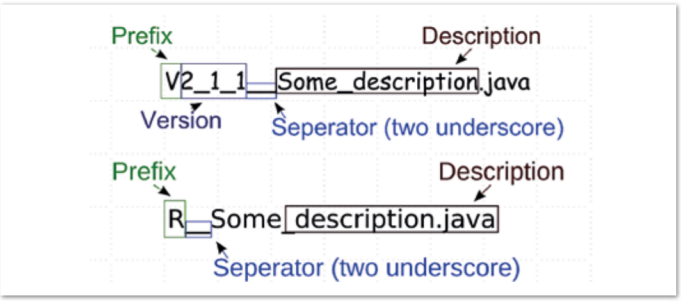

Abbildung 7: Bezeichnung der Java-Skripte

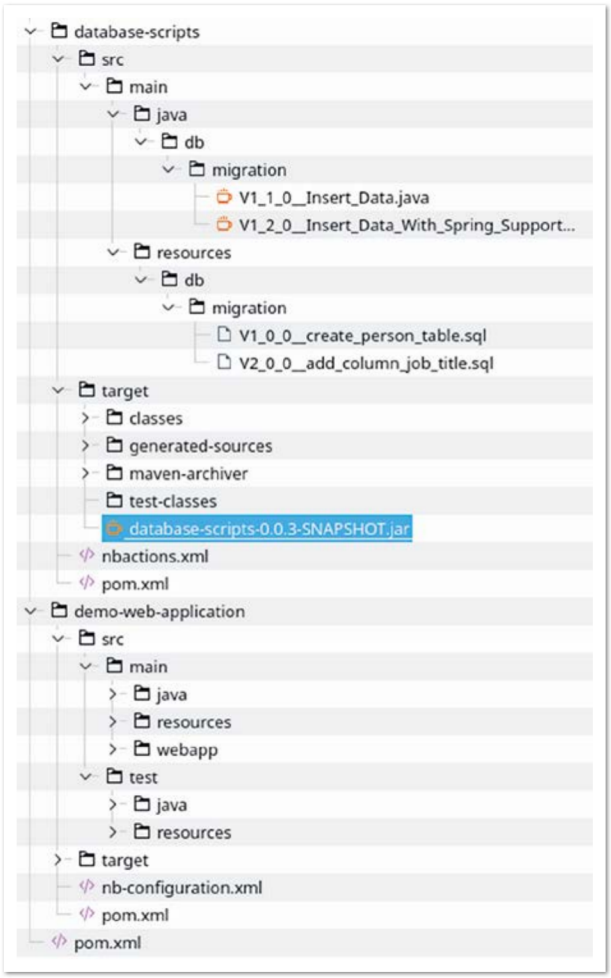

Abbildung 8: Ablageort der Migrationsskripte am Beispiel eines Maven-Projekts

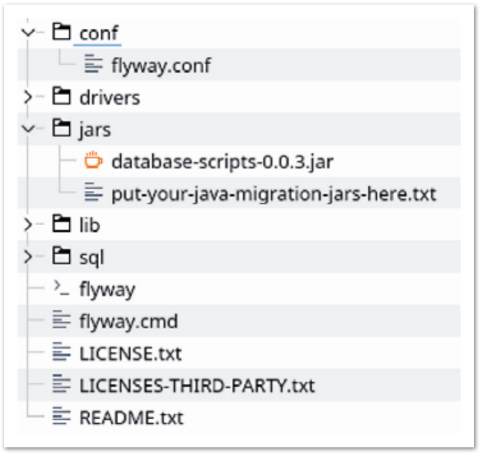

Abbildung 9: Ablageort der Migrationsskripte im Flyway-CLI für die Ausführung

Die Java-basierten Migrationsskripte sind Klassen, die entweder das Flyway-Interface „JdbcMigration“ *(siehe Listing 2)* oder „SpringJdbcMigration“ *(siehe Listing 3)* implementieren. Die Wahl ist abhängig davon, ob das „JdbcTemplate“ von Spring oder lieber Plain JDBC benutzt werden soll.

Typische Anwendungsfälle für die Java-basierten Migrationsskripte sind BLOB- und CLOB-Änderungen oder fortgeschrittene Änderungen an Massendaten (Neuberechnungen, fortgeschrittene Formatänderungen usw.). Der Dateiname der Java-Klassen folgt demselben Schema wie bei den SQL-Skripten, nur dass die Dateiendung „java“ ist *(siehe Abbildung 7)*.

Verwaltung der Migrationsskripte

Die Migrationsskripte werden wie normaler Quelltext behandelt und sollten zusammen mit den anderen Quelltexten des betroffenen Projekts abgelegt sein. Wenn man beispielsweise Maven als Build-Werkzeug benutzt, wird ein eigenes Maven-Modul im Projekt für die Migrationsskripte angelegt. Die Java-basierten Skripte sind unter „src/main/java“ im Package „db.migration“ abgelegt, die SQL-basierten Skripte unter „src/main/resources“ *(siehe Abbildung 8)*.

Die während des Maven-Builds generierte Jar-Datei (im Beispiel „database-scripts-0.0.3.jar“) ist im Flyway-CLI-Werkezeug unter „jars“ abgelegt *(siehe Abbildung 9)*. Mit der passenden Konfiguration unter „conf/flyway.properties“ *(siehe Listing 4)* können die Migrationsskripte per „flyway migrate“-Befehl ausgeführt werden *(siehe Abbildung 10)*.

Testen der Migrationsskripte während eines Builds

Wie auch bei Java-Quelltexten sollen auch die Migrationsskripte während eines Builds getestet werden. Da die Migrationsskripte

```
public class V1_1_0__Insert_Data implements JdbcMigration {

  @Override
  public void migrate(Connection connection) throws Exception {
      try (Statement statement = connection.createStatement()) {
          statement.execute("Insert into person (first_name, last_name) Values ('Alice', 'Bob')");
      }

  }

}
```

Listing 2

```
sparsick@sparsick-ThinkPad-T430s ~/dev/NetBeansProjects/flyway-talk/flyway-4.0.3 $ ./flyway migrate
Flyway 4.0.3 by Boxfuse

Database: jdbc:mysql://192.168.33.10:3306/ (MySQL 5.5)
Successfully validated 3 migrations (execution time 00:00.014s)
Creating schema `flyway_demo` ...
Creating Metadata table: `flyway_demo`.`schema_version`
Current version of schema `flyway_demo`: 0
Migrating schema `flyway_demo` to version 1.0.0 - create person table
Migrating schema `flyway_demo` to version 1.1.0 - Insert Data
Migrating schema `flyway_demo` to version 2.0.0 - add column job title
Successfully applied 3 migrations to schema `flyway_demo` (execution time 00:00.051s).
sparsick@sparsick-ThinkPad-T430s ~/dev/NetBeansProjects/flyway-talk/flyway-4.0.3 $
```

Abbildung 10: Ausführung der Migrationsskripte mit Flyway-CLI

```
 T E S T S
-------------------------------------------------------
Running db.migration.DbMigrationITest
INFO  - ertyClientProviderStrategy - Found docker client settings from environment
INFO  - ckerClientProviderStrategy - Found Docker environment with Environment variables, system properties and defaults. Resolved:
    dockerHost=unix:///var/run/docker.sock
    apiVersion='{UNKNOWN_VERSION}'
    registryUrl='https://index.docker.io/v1/'
    registryUsername='sparsick'
    registryPassword='null'
    registryEmail='null'
    dockerConfig='DefaultDockerClientConfig[dockerHost=unix:///var/run/docker.sock,registryUsername=sparsick,registryPassword=<null>,registryEmail=<null>,regis

INFO  - DockerClientFactory        - Docker host IP address is localhost
INFO  - DockerClientFactory        - Connected to docker:
  Server Version: 17.05.0-ce
  API Version: 1.29
  Operating System: Linux Mint 18.2
  Total Memory: 19511 MB
       ℹ Checking the system...
       ✔ Docker version is newer than 1.6.0
       ✔ Docker environment has more than 2GB free
       ✔ File should be mountable
       ✔ Exposed port is accessible
INFO  - 🐳 [mysql:latest]          - Creating container for image: mysql:latest
INFO  - 🐳 [mysql:latest]          - Starting container with ID: 2668be66c2631e49b5bcb4e180665d223525ec896ea78034326076d5f9063d53
INFO  - 🐳 [mysql:latest]          - Container mysql:latest is starting: 2668be66c2631e49b5bcb4e180665d223525ec896ea78034326076d5f9063d53
INFO  - 🐳 [mysql:latest]          - Waiting for database connection to become available at jdbc:mysql://localhost:32769/test using query 'SELECT 1'
INFO  - 🐳 [mysql:latest]          - Obtained a connection to container (jdbc:mysql://localhost:32769/test)
INFO  - 🐳 [mysql:latest]          - Container mysql:latest started
INFO  - VersionPrinter             - Flyway 4.0.3 by Boxfuse
INFO  - DbSupportFactory           - Database: jdbc:mysql://localhost:32769/test (MySQL 5.7)
INFO  - DbValidate                 - Successfully validated 2 migrations (execution time 00:00.011s)
INFO  - MetaDataTableImpl          - Creating Metadata table: `test`.`schema_version`
INFO  - DbMigrate                  - Current version of schema `test`: << Empty Schema >>
INFO  - DbMigrate                  - Migrating schema `test` to version 1.0.0 - create person table
INFO  - DbMigrate                  - Migrating schema `test` to version 2.0.0 - add column job title
INFO  - DbMigrate                  - Successfully applied 2 migrations to schema `test` (execution time 00:00.133s).
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 13.9 sec
```

Abbildung 11: Ausführung der Tests für Migrationsskripte mit dem Flyway-Java-API und Testcontainers während eines Maven-Builds

```
public class V1_2_0__Insert_Data_With_Spring_Support implements SpringJdbcMigration {

  @Override
  public void migrate(JdbcTemplate jdbcTemplate) throws Exception {
     jdbcTemplate.execute("Insert into person (first_name, last_name) Values ('Charlie', 'Delta')");
  }

}
```

Listing 3

```
flyway.url=jdbc:mysql://192.168.33.10:3306
flyway.user=flyway
flyway.password=flyway
flyway.schemas=flyway_demo
# Unprefixed locations or locations starting with classpath: point to a package on the classpath and may contain
both sql and java-based migrations.
# Locations starting with filesystem: point to a directory on the filesystem and may only contain sql migrations.
flyway.locations=db/migration
```

Listing 4

Datenbank-spezifische SQL-Syntax enthalten können, muss auch eine Datenbank des entsprechenden Datenbank-Anbieters während des Builds vorhanden sein. Da beißen sich zwei Anforderungen an einen Build. Auf der einen Seite soll der Build von externen Anwendungen (wie einer Datenbank) unabhängig sein, zum anderen sollen auch die Migrationsskripte während eines Builds getestet werden, um so früh wie möglich eventuelle Fehler zu erkennen.

Aus diesem Dilemma kann die Java-Bibliothek „Testcontainers" *[3]* weiterhelfen. Sie hilft auf eine einfache Art und Weise, während eines JUnit-Tests Docker-Container *[4]* hochzufahren, und bietet für Datenbanken schon vorgefertigte JUnit-Rules an. Ein Test für die Migrationsskripte sieht dann so aus, dass ein JUnit-Test geschrieben wird, in dem die JUnit-Rule „MySQLContainer" (falls wie im Beispiel mySQL die Datenbank der Wahl ist) vom Test-Container eingebunden wird. Mit dem Flyway-Java-API können dann die Migrationsskripte gegen die Datenbank im Container ausgeführt werden *(siehe Listing 5)*. Dieser Test wird wie gewohnt während des Builds mit ausgeführt *(siehe Abbildung 11)*.

Die Grenzen von Flyway

Es soll nicht verschwiegen werden, dass Flyway auch seine Grenzen hat. Einer der meist genannten Kritikpunkte ist der fehlende Rollback-Mechanismus. Dies war eine bewusste Entscheidung bei Flyway *(siehe [5])*. Die Macher von Flyway empfehlen, bei einer fehlgeschlagenen Migration lieber das Backup der Datenbank wieder einzuspielen.

Wie bei der Vorstellung der SQL-Migrationsskripte erwähnt, unterstützt Flyway Datenbank-spezifisches SQL. Daraus folgt, dass, falls die Anwendung mit unterschiedlichen Datenbank-Anbietern laufen muss, die Migrationsskripte redundant für die jeweilige Datenbank geschrieben werden müssen. Dann ist es eine berechtigte Frage, ob Flyway das richtige Werkzeug für einen ist und ob dieser Anwendungsfall nicht besser mit Liquibase gelöst wird.

Fazit

Dieser Artikel zeigt, welche Grundregeln beachtet und welche Maßnahmen ergriffen werden sollen, wenn die Updates des Datenbank-Schemas kontrolliert und automatisiert ablaufen sollen. Anhand Flyway wird gezeigt, wie ein Werkzeug bei der Umsetzung der Maßnahmen helfen kann.

```
public class DbMigrationITest {
  @Rule
  public MySQLContainer mysqlDb = new MySQLContain-
er();
  @Test
  public void testDbMigrationFromTheScratch(){
     Flyway flyway = new Flyway();
     flyway.setDataSource(mysqlDb.getJdbcUrl(),
mysqlDb.getUsername(), mysqlDb.getPassword());

     flyway.migrate();
  }
}
```

Listing 5

Quellen

[1] Paul M. Duvall, Steve Matyas und Andrew Glover: Continuous Integration, Addison-Wesley (2007)
[2] *http://flywaydb.org*
[3] *https://www.testcontainers.org*
[4] *https://www.docker.com*
[5] *https://flywaydb.org/documentation/faq#downgrade*

Sandra Parsick
mail@sandra-parsick.de

Sandra Parsick, geb. Kosmalla, ist als freiberufliche Software-Entwicklerin und Consultant im Java-Umfeld tätig. Seit dem Jahr 2008 beschäftigt sie sich mit agiler Software-Entwicklung in verschiedenen Rollen. Ihre Schwerpunkte liegen im Bereich der Enterprise-Anwendungen, agilen Methoden, Software Craftsmanship und in der Automatisierung von Software-Entwicklungsprozessen. In ihrer Freizeit engagiert sie sich in der Softwerkskammer Ruhrgebiet.