03-2014 | Herbst | www.ijug.eu

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

Aus der Community – für die Community

Java aktuell

Java im Mittelpunkt

Aktuell
NoSQL für Java

Performance
Java-Tuning

Pimp my Jenkins

Logging
Apache Log4j 2.0

Sonderdruck

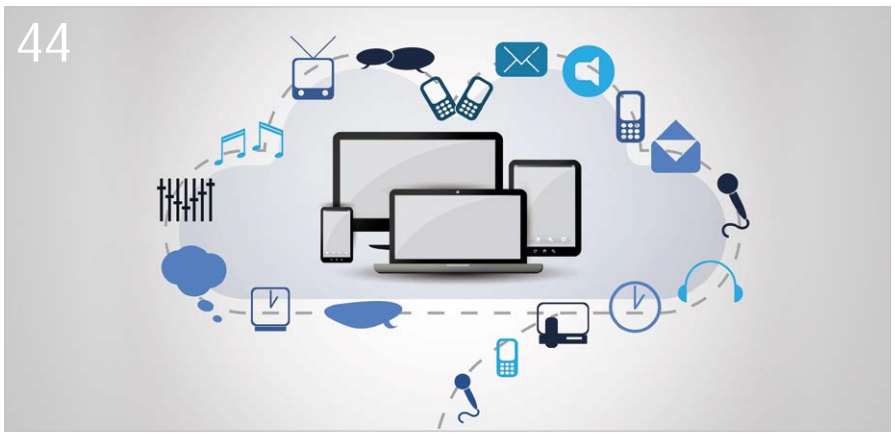
D: 4,90 EUR A: 5,60 EUR CH: 9,80 CHF Benelux: 5,80 EUR ISSN 2191-6977

4 191978 304903 02

iJUG Verbund

JavaLandConf 2014 Twitterwall

JavaFX– das neue Gesicht für Java-Anwendungen



Connectivity-as-a-Service für Cloud-basierte Datenquellen

Unbekannte Kostbarkeiten des SDK

die Entwicklungs- und Betriebsabteilungen auf die gemeinsame Definition, Implementierung und laufende Verbesserung der Build-, Test- und Deployment-Prozesse konzentrieren. Und im Idealfall bedingt die Entwicklung einer neuen fachlichen Anwendung für den Betrieb lediglich eine höhere Auslastung der vorhandenen Infrastruktur ohne nennenswerte Mehrarbeit.

Quellen

[1] http://de.wikipedia.org/wiki/Parkinsonsche_Gesetze
[2a] https://appfactorypreview.wso2.com
[2b] http://wso2.com/cloud/app-factory
[3] Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation; Humble and Farley, Addison-Wesley 2010; ISBN 0-321-60191-2
[4] Suse Cloud: https://www.suse.com/de-de/products/suse-cloud
[5] Prinzipien des API-Managements; T. Unger und J. Traunecker, Java aktuell, Ausgabe 1/2014

Jochen Traunecker
jochen.traunecker@gridsolut.de

Database-DevOps mit MySQL, Hudson, Gradle, Maven und Git

Michael Hüttermann

DevOps, ein Kofferwort aus Development (Entwicklung) und Operations (Betrieb), beschreibt die optimierte Zusammenarbeit eben dieser Funktionen in IT-Unternehmen und -Projekten. In der Software-Entwicklung liegen Datenbanken häufig auf einem kritischen Pfad. Dieser Artikel liefert eine Definition von DevOps und erläutert, wie Datenbank-DevOps mit Konzepten und Werkzeugen konkret aussehen kann.

Konflikte zwischen Entwicklung und Betrieb können viele Ursachen haben, etwa unterschiedliche Ziele (mehr Veränderungen in kurzer Zeit für die Entwicklung, wenig Veränderungen und Stabilität in der Produktion für den Betrieb), unterschiedliche Prozesse und Konzepte (pragmatische Ansätze für Entwicklung, mehr Fokus auf Rückverfolgbarkeit für den Betrieb) und unterschiedliche Werkzeuge (Entwicklungswerkzeuge für Entwicklung, produktionstaugliche Ansätze für den Betrieb). Diese Unterschiede reißen oft Lücken zwischen den Abteilungen beziehungsweise begünstigen Silos, die als Nächstes erläutert werden.

Verschiedene Teams

In den traditionellen Konstellationen umfasst der Begriff „Entwicklung" die Programmierer im Team. Tester sind traditionell dedizierte Projekt-Rollen, die einen Großteil ihrer Aktivitäten erst starten, nachdem die Programmierer ihre Arbeit beendet haben. Dank der agilen Software-Entwicklung überlagern sich diese Rollen und deren Aktivitäten zunehmend, sodass mittlerweile ein inkrementell/iterativer Ansatz mit einer Verzahnung von Programmierung und Test mehr die Regel als die Ausnahme ist. Der „Betrieb" umfasst Rollen wie Datenbank-, System- und Netzwerk-Administratoren sowie weitere Administratoren, die Server und Systeme einrichten und warten. Der Betrieb begleitet die Änderungen auf der sogenannten „letzten Meile", bis diese dem Benutzer in der Produktion zur Verfügung gestellt sind. In einer barrierelastigen Umgebung formen Entwicklung und Betrieb lokal optimierte Gruppen mit jeweils eigenen Zielen, Prozessen und Werkzeugen (siehe Abbildung 1).

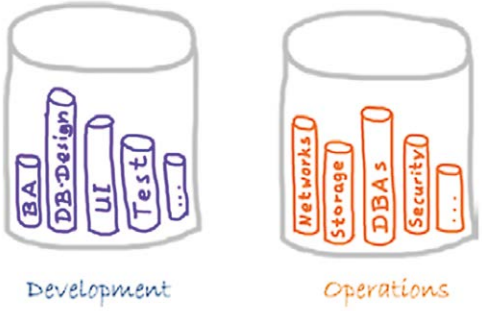


Abbildung 1: Entwicklung und Betrieb, zwei unterschiedliche Teams

Unterschiedliche Ansätze

Es ist nicht immer einfach, die richtige Software (Effektivität) richtig (Effizienz) zu entwickeln. Insbesondere die Entwicklung von Datenbanken ist recht häufig eine Herausforderung. Warum? Traditionell stellt die Entwicklung ihre Änderungen via Check-in in ein Versionskontrollsystem den Kollegen und anderen Teams zur Verfügung. Fachlicher Anwendungscode wird lokal getestet und verfolgt, ohne dass diese Arbeiten mit der Arbeit von Kollegen direkt interferieren würden.

Deployments auf Zielumgebungen werden häufig über einen zentralen Build-Server kanalisiert und regelmäßig durchgeführt. Auf der anderen Seite fokussiert sich die Datenbank-Entwicklung häufig auf eine zentrale Ressource, auch wenn Entwickler über eigene Datenbanken oder eigene Schemata auf einer zentralen Datenbank verfügen. Ferner ist Datenbank-Entwicklung kein Prozess von schlichtem „Copy & Paste", vor und zurück. So lässt sich zum Beispiel eine Datenbank-Tabelle nicht einfach löschen und mit einer neuen Struktur wiederherstellen.

Selten sind zwei Status tatsächlich gleich, da entweder die Quelle oder das Ziel von früheren Installationen und Entwicklungen geändert wurde. Als Ergebnis ergeben sich aus all diesen Aspekten große Unterschiede zwischen Entwicklung der eigentlichen Fachanwendung sowie der Datenbank beziehungsweise zwischen Entwicklung und Betrieb.

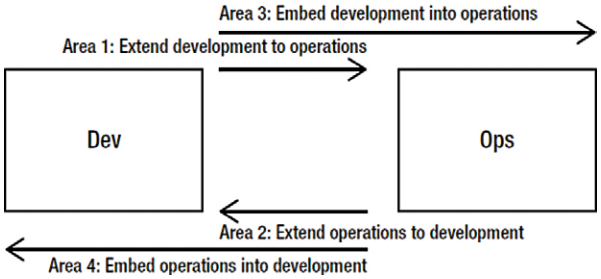


Abbildung 2: Die DevOps-Matrix

DevOps in aller Kürze

DevOps beschreibt Praktiken, die den Softwarebereitstellungsprozess ganzheitlich betrachten und optimieren. DevOps möchte Ziele, Prozesse und Werkzeuge von Entwicklung und Betrieb zueinander ausrichten und damit die Zusammenarbeit verbessern. Durch einen verbesserten Fluss soll Software schneller und in besserer Qualität vom Programmierer zum Nutzer gelangen. DevOps umfasst zahlreiche Aktivitäten und Aspekte wie die folgenden:

- *Culture*
  Software wird von Menschen für Menschen gemacht. Jedes Projekt und jedes Unternehmen hat eine eigene Kultur.
- *Automation*
  Automatisierung ist essenziell, um schnelle Rückkopplungen zu erhalten.
- *Measurement*
  DevOps wählt bestimmte Ansätze zur Definition und Messung von Qualität.
- *Sharing*
  Eine Kollaboration durch Teilen von Wissen und Erfahrungen ist unumgänglich.

DevOps geht von einem umfassenden Ansatz aus, der den kompletten Kernprozess begleitet. Ein erster Schritt in diese Richtung kann ein Value-Stream-Mapping liefern, um die Wertschöpfungskette ganzheitlich zu erheben, noch besser zu verstehen und Flaschenhälse zu identifizieren. Aufbauend darauf können auch kausale Abhängigkeiten erkannt und Feedbackschleifen eingeflochten werden. Dabei ist allen Teilnehmern genug Platz zum Experimentieren einzuräumen. Damit ist kein sinnfreies Basteln gemeint, sondern das zielgerichtete, doch experimentelle, explorative Arbeiten mitsamt Pufferzeiten, um auf Änderungen auch zukünftig möglichst flexibel eingehen zu können und Fallstricken bei der Automation zu begegnen, siehe dazu auch Kapitel 3 in „DevOps for Developers", Hüttermann, Apress, 2012.

```
michael@michael-
VirtualBox:~/talk/project/devops/src/main/resources/db/migration$
ls -la
total 16
drwxrwxr-x 2 michael michael 4096 Sep 22 11:16 .
drwxrwxr-x 3 michael michael 4096 Sep 22 11:16 ..
-rw-rw-r-- 1 michael michael 112 Sep 18 07:59 V1__Create_person_table.sql
-rw-rw-r-- 1 michael michael 149 Sep 18 07:28 V2__Insert_persons.sql
```

Listing 1

```
create table PERSON (
ID int not null auto_increment,
NAME varchar(80) not null,
primary key (ID)
);
```

Listing 2

```
INSERT INTO PERSON (NAME) VALUES ('Peter Meyer');
INSERT INTO PERSON (NAME) VALUES ('Peter Bonnd');
INSERT INTO PERSON (NAME) VALUES ('Klara Korn');
```

Listing 3

```
<profile>
   <id>db</id>
   <build>
      <plugins>
         <plugin>
            <groupId>com.googlecode.flyway</groupId>
            <artifactId>flyway-maven-plugin</artifactId>
            <version>2.1.1</version>
            <configuration>
               <user>fly</user>
               <password>way</password>
               <driver>com.mysql.jdbc.Driver</driver>
               <url>jdbc:mysql://localhost:3306/mydb</url>
               <baseDir>db/migration</baseDir>
            </configuration>
         </plugin>
      </plugins>
   </build>
   <dependencies>
      <dependency>
         <groupId>mysql</groupId>
         <artifactId>mysql-connector-java</artifactId>
         <version>5.1.25</version>
      </dependency>
   </dependencies>
</profile>
```

Listing 4

```
[INFO] --- flyway-maven-plugin:2.1.1:migrate (default-cli) @ de
vops ---
[INFO] Creating Metadata table: `mydb`.`schema_version`
[INFO] Current version of schema `mydb`: << Empty Schema >>
[INFO] Migrating schema `mydb` to version 1
[INFO] Migrating schema `mydb` to version 2
[INFO] Successfully applied 2 migrations to schema `mydb` (execu-
tion time
00:00.234s).
```

Listing 5

```
mysql> show tables;
+----------------+
| Tables_in_mydb |
+----------------+
| PERSON         |
| schema_version |
+----------------+
```

Listing 6

```
mysql> select * from PERSON;
+----+-------------+
| ID | NAME        |
+----+-------------+
| 1  | Peter Meyer |
| 2  | Peter Bonnd |
| 3  | Klara Korn  |
+----+-------------+
3 rows in set (0.00 sec)
```

Listing 7

Bei einer Implementierung von Datenbank-DevOps kann man die DevOps-Matrix heranziehen (siehe Abbildung 2). Bereich 1 der Matrix beschreibt die Ausweitung agiler Verfahren von der Entwicklung in Richtung Betrieb. Ein häufiger Anwendungsfall im Datenbank-Kontext ist das Pflegen von Datenbank-Skripten über ein Versionskontrollsystem und die Verwendung von Werkzeugen wie Flyway, etwas später in diesem Artikel weiter thematisiert. Bereich 2 ist die Ausweitung von Praktiken in die andere Richtung, also vom Betrieb in Richtung Entwicklung. Für Datenbank-DevOps kann das bedeuten, Infos über Live-Traffic oder Datenbank-Ressourcenkonflikte auch der Entwicklung zur Verfügung zu stellen.

Bereich 3 bettet die Entwicklung in den Betrieb ein. Damit ist keine organisatorische Umverteilung gemeint, sondern vielmehr die Ausrichtung von Zielen oder die Aufnahme von nicht-funktionalen Anforderungen sehr früh in die Entwicklung. Gemeinsame Ziele können sein:

- 80 Prozent der Datenbank-Abfragen liefern Ergebnisse auf dem Bildschirm in weniger als zwei Sekunden (ein gemeinsames Performance-Ziel)
- Kein Gebrauch von Technologien, die einen späteren Wechsel auf andere Technologien ausschließen oder erschweren, wie Datenbank-proprietäre SQL-Routinen (ein gemeinsames Portabilitäts-Ziel)
- 250 Milliarden Datensätze können mit der gegebenen Hardware gespeichert werden, bei gleichzeitiger Einhaltung der Performance-Ziele (ein gemeinsames Kapazitäts-Ziel)
- Automatisierte Tests existieren für alle Komponenten einschließlich Infrastruktur-Code (ein gemeinsames Wartbarkeits-Ziel)

Schließich beschreibt Bereich 4 die Einbettung des Betriebs in die Entwicklung. Dies kann zum Beispiel durch die Bereitstellung von Informationen innerhalb der Entwicklung erreicht werden, ohne bei jedem Informationszugriff auf einen DBA angewiesen zu sein, oder die Aufnahme eines Betrieblers zu 10 Prozent seiner Kapazität in ein Scrum-Team der Entwicklung. Nachdem wir nun konkrete Beispiele anhand der DevOps-Matrix durchleuchtet haben, vertiefen wir nachfolgend die Bereiche, allen voran Bereich 1, mit weiteren technischen Details.

Datenbank-DevOps

Um die Zusammenarbeit zwischen Entwicklung und Betrieb zu optimieren, verbindet eine robuste Lösung für Datenbank-DevOps traditionelle Lösungen des Betriebs mit Ansätzen wie Datenbank-Versionskontrolle, Continuous Integration und Automatisierung. Die folgenden Muster können helfen, Datenbank-DevOps einzuführen:

- Legen Sie alle Code-und Datenbank-Elemente (sowie alle Änderungssätze) in die Versionskontrolle.

```
+----------------+----------------------------+---------------------+---------+
| Version        | Description                | Installed on        | State   |
+----------------+----------------------------+---------------------+---------+
| 1              | Create person table        | 2013-09-19 20:52:32 | Success |
| 2              | Insert persons             | 2013-09-19 20:52:32 | Success |
| 3              | InsertUpdate persons       | 2013-09-19 20:55:52 | Success |
+----------------+----------------------------+---------------------+---------+
```

Listing 8

```
apply plugin: 'java'
apply plugin: 'flyway'

flyway {
   user = 'fly'
   password = 'way'
   driver = 'com.mysql.jdbc.Driver'
   url = 'jdbc:mysql://localhost:3306/mydb'
   baseDir = 'db/migration'
}

buildscript {
   repositories {
      mavenCentral()
   }

dependencies {
   classpath "com.googlecode.flyway:flyway-gradle-plugin:2.2"
   classpath "mysql:mysql-connector-java:5.1.25"
   }
}

repositories {
   mavenCentral()
}

dependencies {
   testCompile 'junit:junit:4.11'
}
```

Listing 9

```
UPDATE PERSON SET NAME='Peter Bond' WHERE ID=2;
DROP PROCEDURE IF EXISTS AddPerson;
delimiter //
CREATE PROCEDURE AddPerson (IN myvalue VARCHAR(80))
BEGIN
INSERT INTO PERSON (NAME) VALUES (myvalue);
END //
delimiter ;
CALL AddPerson(,Donald Luck');
```

Listing 10

- Erstellen Sie SQL (Structured Query Language)-Skripte, die angewendet werden müssen, um zur nächsten Version migrieren zu können (Roll-Forward). Prüfen Sie, ob Sie gegebenenfalls Roll-Backward vollständig wegoptimieren können.
- Erstellen Sie für jede logische Einheit von Änderungen (Change-Sets) eine Datei, die die Änderungsskripte beinhaltet. Geben Sie der Datei einen eindeutigen Namen einschließlich einer Nummerierung, die sich kardinal skalieren lässt.
- Erstellen Sie Baselines, frieren Sie alle Konfigurationselemente ein, einschließlich Anwendung, Middleware und Datenbank.
- Setzen Sie bei Migrationen auf diese Baselines auf. In Fällen einer vollständigen Installation wenden Sie alle Change-Sets aufbauend auf der Baseline an. Bei inkrementeller Installation führen Sie ausschließlich die jeweils fehlenden Change-Sets aus.
- Platzieren Sie Ihre Datenbank-Migrationsskripte in entsprechende Verzeichnisse.
- Berücksichtigen Sie Maßnahmen zur Überwachung, um „Mean Time To Repair" (MTTR) und „Mean Time To Detect" (MTTD) zu minimieren, sowie Smoke-Tests.
- Speichern Sie die Datenbank-Version. Ein Ansatz ist, eine dedizierte Datenbank-Tabelle fortzuschreiben, die Metadaten beinhaltet, insbesondere darüber, in welchem Zustand sich die Datenbank aktuell befindet.

Eine exemplarische Werkzeugkette

Sie können selbst eine Lösung entwickeln und die weiter oben aufgeführten Rezepte vollständig implementieren oder Sie nutzen ergänzend ein Werkzeug, das dabei hilft, diese Konzepte umzusetzen. Ein solches Werkzeug ist Flyway (siehe http://flywaydb.org). Im Folgenden werden wir ein Beispiel skizzieren, in dem wir Werkzeuge wie Flyway, Maven, Gradle, Git und Hudson miteinander integrieren. Beginnen wir mit einem Blick in den Verzeichnisbaum, in dem die Migrationsskripte liegen (siehe Listing 1).

Im Moment liegen zwei Dateien im Verzeichnis. Eine Datei beinhaltet ein Change-Set, also einen Satz von nativen SQL-Statements, die auf eine Datenbank angewendet werden sollen. Auch das ist DevOps: Definieren Sie Change-Sets ausschließlich in nativem SQL und kompilieren Sie beispielsweise diese SQL-Statements nicht in eine höhere Programmiersprache hinein. Die erste Datei ist „V1__Create_person_table.sql". Es handelt sich um eine

DDL-Datei (Data Definition Language); Listing 2 zeigt ihren Inhalt.

Die zweite Datei ist eine DML-Datei (Data Manipulating Language). Die Datei „V2__Insert_persons.sql" besteht aus drei Anweisungen zum Anlegen von drei Sätzen in der zuvor erstellten Tabelle (siehe Listing 3).

Da wir automatisieren möchten, schauen wir zunächst auf eine Maven-POM, die die Migration anstoßen soll (Weitere Infos zu Maven siehe http://maven.apache.org). Die POM ist der Ort, an dem wir beispielsweise Datenbank-Verbindungsparameter hinterlegen können, auch parametrisiert. Achtung: Abhängig von konkreten Anforderungen ist natürlich nicht immer ein Build-Skript der beste Ort, um Migrationsskripte anzustoßen. In unserem Fall ist die Logik in einem Maven-Profil hinterlegt. Listing 4 zeigt den relevanten Ausschnitt.

Flyway arbeitet mit Classpath-Scanning. Bei der Nutzung aus Maven heraus müssen Migrationsskripte auf das Ziel kopiert werden, also zum Beispiel der Maven-Zielordner „target". Der Maven-Anruf kann dann so aussehen: „clean install flyway:migrate –Pdb". Um den Aufruf noch komfortabler zu gestalten, von informativer Visualisierung zu profitieren oder auch eine umfangreiche Delivery-Pipeline aufzubauen, integrieren wir auch Hudson, den Build Server (siehe http://hudson-ci.org). Ein automatischer Aufruf des Maven-Profils via Hudson kann dann folgende Konsolenausgabe zur Folge haben (siehe Listing 5).

```
git add V3__InsertUpdate_persons.sql
git commit -m "change" .
git push
```

Listing 11

```
[INFO]
[INFO] --- flyway-maven-plugin:2.1.1:migrate (default-cli) @
devops ---
[INFO] Current version of schema `mydb`: 2
[INFO] Migrating schema `mydb` to version 3
[INFO] Successfully applied 1 migration to schema `mydb` (execu-
tion time
00:00.069s).
```

Listing 12

In unserem Fall war die Datenbank zunächst leer, es wurden also noch keine Migrationen angewendet. Nach etwas Bootstrapping (Erstellen von Meta-Informationen) wendet Flyway die beiden Migrationen auf die Datenbank an, da wir zwei SQL-Dateien in das Verzeichnis gelegt haben. Listing 6 zeigt, was ein schneller Gegen-Check auf der Datenbank liefert.

Werfen wir nun einen kleinen Blick in die Tabelle „Person". Die Tabelle hat drei Sätze, da die beiden Migrationsskripte die Tabelle anlegen und drei Sätze einfügen (siehe Listing 7).

Auch das verlief erfolgreich. Flyway bietet die Möglichkeit, durch eine „flywayInfo" den Stand der Migrationen zu zeigen. An

```
mysql> select * from PERSON;
+----+-------------+
| ID | NAME        |
+----+-------------+
| 1  | Peter Meyer |
| 2  | Peter Bond  |
| 3  | Klara Korn  |
| 4  | Donald Luck |
+----+-------------+
4 rows in set (0.00 sec)
```

Listing 13

das Ergebnis kommen wir beispielsweise auch durch Gradle (siehe http://www.gradle.org) und den Aufruf „gradle flywayInfo" (siehe Listing 8).

Listing 9 zeigt die Gradle-Build-Datei, die funktional vergleichbar ist mit der

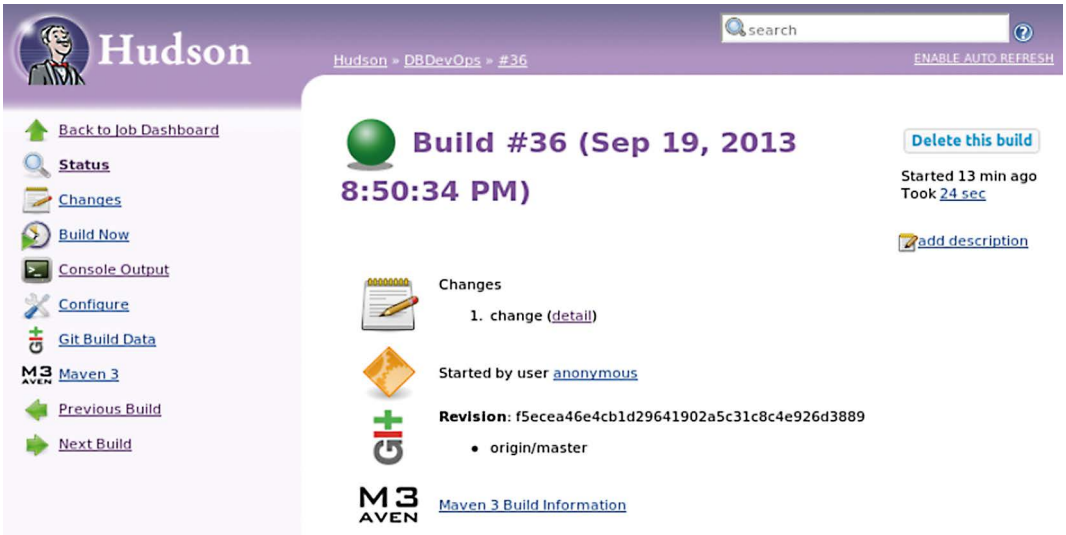


Abbildung 3: Hudson lauscht auf Änderungen in Git und löst Datenbank-Konvertierungen aus

Maven-POM. Die Datei ist ebenfalls Teil des Quellcodes im Versionskontrollsystem.

Wir nehmen nun die nächste Migration in Angriff. Eine weitere Datei, unser drittes Migrationsskript mit dem Namen „V3__InsertUpdate_persons.sql", enthält ein Update auf einen vorhandenen Datensatz sowie die Erstellung einer Stored Procedure. Diese wird dann schließlich genutzt, um einen weiteren Datensatz einzufügen (siehe Listing 10).

Wir können die neue Datei nun in das Git-Repository einspeisen und so eine Datenbank-Migration auf einem Testsystem anstoßen (siehe Listing 11). Hudson erkennt die Änderungen in Git und baut das Projekt (siehe Abbildung 3).

Die Werkzeugkette liest den aktuellen Zustand der Datenbank und erkennt, dass nun ein neues Migrationsskript vorliegt, das angewendet werden soll. Die aktuelle Version der Datenbank ist „2". Die neu verfügbare Migration hat die Nummer „3" (siehe Listing 12).

Wir haben anschließend einen vierten Datensatz in unserer Tabelle „PERSON" sowie eine aktualisierte Spalte, offenbar ein Fix eines zuvor eingeführten Rechtschreibfehlers. Der neue Tabelleneintrag wurde durch den Aufruf der frisch erstellten Stored Procedure eingefügt. Wir schauen in die Tabelle, um uns vom ordnungsgemäßen Ablauf zu überzeugen (siehe Listing 13). Nun ist die Datenbank in den gewünschten Zielzustand überführt.

Fazit

Dieser Artikel führte in DevOps ein, eine moderne Herangehensweise, die sich eine Angleichung von Zielen, Prozessen und Werkzeugen zwischen Entwicklung und Betrieb zum Ziel gesetzt hat. Aufbauend auf einem konkreten Beispiel haben wir eine kleine Rundreise durchgeführt durch relevante Praktiken und Werkzeuge.

Michael Hüttermann
michael@huettermann.net

Java-Champion Michael Hüttermann (http://huettermann.net) ist freiberuflicher Delivery Engineer und Experte für DevOps, Continuous Delivery und SCM/ALM. Er schrieb die ersten Bücher zu „Agile ALM" (Manning, 2011) und DevOps („DevOps für Entwickler", Apress, 2012).

Entweder ... oder Fehler

Heiner Kücker, Freiberufler

Es ist nicht sicher, ob Scala eines Tages Java ablösen wird, aber in Scala und seinen Libs sind Pattern/Idiome manifestiert, die der Alltags-Java-Programmierung durchaus guttun würden.

Der Autor hatte mal ein Problem im Programm eines Kollegen übernommen. Darin gab es eine Suchfunktion, die ein recht merkwürdiges Verhalten hatte. Der Suche wurde eine Liste von Nummern, beispielsweise Artikel-Nummern, übergeben. War diese Liste „null", wurde im SQL-Select-Statement keine „WHERE IN"-Klausel erzeugt. War die Liste leer, wurde die Suche nicht ausgeführt, was aber kein Problem war, da die Suche Teil eines SQL-Union-Befehls war und im weiteren Java-Code noch andere Objekte zur Ergebnisliste hinzugefügt wurden. War die Liste nicht leer, wurde eine „WHERE IN"-Klausel erzeugt. Nochmal ganz langsam:

- *Liste null*
  Alle Werte erlaubt
- *Liste leer*
  Keine Werte erlaubt
- *Liste mit Werten*
  Werte erlaubt

„null" ist also eine magische Nummer für das Ignorieren des Such-Parameters. Die zweite Variante, die leere Liste, würde ungültiges SQL erzeugen, das aber sowieso keine Ergebnisse liefern würde. Die dritte Variante, die „nicht leere" Liste mit Werten, wird in SQL als Selektions-Kriterium eingebaut. Logisch, dass hier nichts kommentiert war.

Das Ganze wurde noch verschleiert durch ein Parameter-Objekt mit Gettern und Settern, in dessen Member-Initialisierung eine leere Liste initialisiert wurde, die irgendwo im Code kommentarlos mit „null" überschrieben wurde, also ein globaler veränderlicher Zustand. Keine Ahnung, warum Eclipse bei der Referenz-Suche von Membern nicht die Getter und Setter mit einbezieht; für den Autor noch ein Grund, unsinnige Getter und Setter zu entfernen.

Parameter-Objekt – auch so ein Anti-Pattern (wie Observer, sollte vielleicht „Obscurer" genannt werden), wenn es zu einer Krabbelkiste von irgendwo einmal benutzten Membern ausartet, die vielleicht