Ein sauber automatisierter Auslieferungsprozess, Unit-Tests und eine kontinuierliche Integration steigern die Qualität der Software deutlich. Im Gegensatz zu anderen Bereichen finden diese Konzepte bei der datenbanknahen Software-Entwicklung keine schnelle Verbreitung. Der Artikel zeigt, welche Lösungsansätze heute schon existieren, aber auch, welche Herausforderungen technischer und organisatorischer Art zu bewältigen sind.

Continuous Integration mit PL/SQL

Andrej Pashchenko, Trivadis AG

Continuous Integration (CI) ist eine Best Practice, die nicht neu ist und in den letzten Jahren vor allem von sogenannten „agilen Softwareentwicklungsmethoden" wie Extreme Programming oder Scrum popularisiert wurde. Die Idee ist einfach: Statt die Integration als nachgelagerte, schwer planbare Phase in einem Software-Entwicklungsprozess zu sehen, beginnt man früh mit der Integration und integriert kontinuierlich. Die Vorteile liegen auf der Hand: Es wird immer „stückchenweise" integriert und damit eine gute Planbarkeit erreicht; Bugs und Integrationsprobleme sind viel einfacher und kostengünstiger zu beheben; die letzte integrierte Version der Software ist für Qualitätssicherungs- oder Demo-Zwecke immer vorhanden. Diese Vorgehensweise lässt die Qualität der Software steigen und man kann schnell und agil auf sich ständig ändernde Fachanforderungen reagieren.

Während Continuous Integration beispielsweise im Bereich der Java-Entwicklung bereits alltäglich ist und eine Reihe erfolgreich eingesetzter Tools vorweisen kann, ist der Bereich der datenbanknahen Software-Entwicklung, bei der ein (Groß-)Teil der Geschäftslogik in PL/SQL implementiert ist, etwas zurückgeblieben.

Big Picture

Abbildung 1 zeigt grob das Zusammenspiel aller CI-Komponenten. Ein Entwickler beginnt, an einer Aufgabe zu arbeiten. Er aktualisiert seine Arbeitskopie und holt sich den aktuellen Quellcode aus dem Versionskontrollsystem (VCS). Unter anderem passt er das Datenmodell an, macht Änderungen am PL/SQL-Code, entwickelt neue Tests für seinen Code oder passt die bestehenden an und erstellt Auslieferungsskripte für alle Änderungen. Wenn er damit fertig ist, checkt er die Ergebnisse seiner Arbeit in VCS ein. Eine Continuous-Integration-Engine überwacht das VCS-Repository und stellt fest, dass neuer Code eingecheckt worden ist. Dies löst einen Integrations-Build aus. Alternativ dazu kann man diesen auch in regelmäßigen Zeitabschnitten (etwa als „nightly build") auslösen. Die CI-Engine holt sich den Quellcode aus dem VCS-Repository und lässt ein sogenanntes „Build-Tool" die Anwendung in das Zielsystem ausliefern und testen. Anschließend berichtet die CI-Engine über den Status des Builds und kann bei Erfolg den Stand im VCS-Repository entsprechend markieren.

Voraussetzungen und Herausforderungen

Aus dem Bild lassen sich bereits einige Voraussetzungen ablesen. Um den CI-Prozess aufzustellen, braucht man natürlich eine Software-Infrastruktur. Die Tools und Systeme, die wir uns später noch genauer anschauen, sind kostenfrei und einfach in Installation und Bedienung. Außerdem ist es sehr wahrscheinlich, dass in größeren gemischten Teams und Projekten diese Infrastruktur bereits besteht und beispielsweise von Java-Entwicklern genutzt wird. Aus rein technischer Sicht wäre es also nicht so schwer, die Voraussetzungen zu erfüllen. Viel wichtiger ist es aus Sicht des Autors, dass man zum Teil kardinale Prozess-Umstellungen benötigt. Wenn man es nicht schafft, diese konsequent durchzuziehen, gibt es keine Erfolgsaussichten für den CI-Prozess. Er wird dann nur noch als Produktivitätshindernis angesehen.

Die wichtigste Voraussetzung ist, dass man einen vollautomatischen Auslieferungsprozess hat, der auch Unit-Tests beinhaltet, denn nach jedem Commit im VCS oder zumindest täglich läuft die Auslieferung. Offensichtlich genügt es

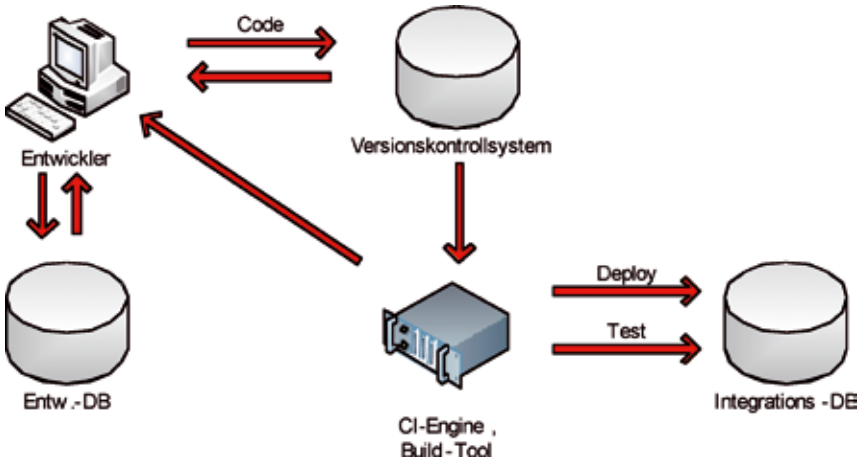

Abbildung 1: Continuous Integration

COMPUTERWOCHE

Jetzt auch für das iPad

* multimedial aufbereitet
* schon freitags verfügbar
* aktuelle Ausgabe nur 3,99 €

Erhältlich im App Store

www.computerwoche.de/ipad

IDG BUSINESS MEDIA GMBH

COMPUTERWOCHE erscheint im Verlag IDG Business Media GmbH, Lyonel-Feininger-Str. 26, 80807 München, Registergericht München, HRB 99187, Geschäftsführer: York von Heimburg. Die Kundenbetreuung erfolgt durch den COMPUTERWOCHE Kundenservice, ZENIT Pressevertrieb GmbH, Postfach 810580, 70522 Stuttgart, Geschäftsführer: Joachim John, Tel.: 0180 5 72 7252-276*, E-Mail: shop@computerwoche.de
(*aus dem dt. Festnetz nur 0,14 Euro pro Minute, Mobilfunkpreise max. 0,42 Euro pro Minute)

C-011-222

nicht, einfach den PL/SQL-Code in das Integrationssystem auszuliefern und zu testen. Häufig steht eine Veränderung im Code in einem Zusammenhang mit Änderungen an Datenbank-Objekten wie Tabellen, Views etc. Damit der PL/SQL-Code überhaupt erst kompilierbar ist, muss man solche Änderungen mitausliefern. Genau hier liegt der größte Unterschied zum klassischen Prozess, bei dem es vor allem um das Kompilieren der Binaries aus dem Quellcode geht. Beispielsweise kann man eine produktiv genutzte Tabelle nur durch „ALTER DDL"-Befehle ändern. Auch eine Datenmigration ist im Normalfall einzubeziehen.

Man kann argumentieren, dass es im Testsystem möglich ist, die ganzen Datenbank-Schemata immer neu anzulegen. Der Nachteil dieses Verfahrens ist aber, dass man für das Testen ein völlig anderes Auslieferungsverfahren hätte, als man gezwungenermaßen beim Ausliefern in die Produktion einsetzen würde. Es wäre aber ein enormer Mehrwert des CI-Prozesses, wenn sich dadurch nicht nur die Funktionstüchtigkeit des PL/SQL-Codes nach jeder Änderung bestätigen ließe, sondern auch der Auslieferungsprozess an sich immer mit getestet würde. Aus diesem Grund muss man auch dafür sorgen, dass die Integrations-Datenbank bei jedem Build zuerst automatisch auf den Stand gebracht wird, der dem Zielsystem entspricht.

Um einen Auslieferungsprozess zu unterstützen, muss eine zentrale Stelle vorhanden sein, an der alle zum Projekt gehörenden Quellcodedateien oder Artefakte, wie man sie im Bereich von Versionskontrollsystemen nennt, gespeichert werden. Diese Stelle, das Repository des Versionskontrollsystems, wird zur absoluten Wahrheit: Alles, was zum Aufbau der Anwendung nötig ist, muss dort vorhanden sein. Sehr wahrscheinlich, dass diese Anforderung in den meisten Projekten bereits erfüllt ist. Das Versäumnis, Datenbankschema-Definitionen als Teil einer Applikation unter Versionskontrolle zu bringen, tritt immer seltener auf.

Auch der Entwicklungsprozess an sich muss häufig angepasst werden. Damit Continuous Integration funktioniert, müssen einzelne Entwickler es sich angewöhnen, ihre Arbeit in kleinere, in sich abgeschlossene Einheiten zu teilen, die man schnell beenden und für das Testen bereitstellen kann. Diese Einheiten sind auch erst dann abgeschlossen, wenn Testfälle definiert und implementiert sind. An dieser Stelle kann man den sogenannten „Test Driven Development (TDD)"-Ansatz anwenden: Man fängt die Implementierung mit Unit-Tests an und beginnt erst dann mit eigentlicher Funktionalität.

Tools

Es gibt eine Reihe von Tools, die man für diese Zwecke einsetzen kann. Wir stellen die Auswahl kurz vor, ohne auf die Installation und Konfiguration ausführlich einzugehen: Sie sind ziemlich trivial. Als Versionskontrollsystem kommen am häufigsten CVS oder sein Nachfolger Subversion zum Einsatz. Beide sind Open-Source-Produkte, haben eine Vielzahl von komfortablen Client-Anwendungen wie beispielsweise TortoiseSVN und bieten eine gute Anbindung an Entwicklungswerkzeuge wie TOAD, SQL-Developer, JDeveloper etc.

Als Nächstes braucht man ein Framework, um die Unit-Tests für den PL/SQL-Code zu entwickeln. Die Wahl fällt hier auf utPLSQL – ein Open-Source-Framework, das Steven Feuerstein ursprünglich entwickelt hat. Zwar wird dieses Framework in letzter Zeit anscheinend nicht weiterentwickelt, es bietet aber eine solide Grundfunktionalität an und ist vor allem derzeit am einfachsten an andere Komponenten wie eine CI-Engine anzubinden. Der Nachteil hierbei ist, dass man einen hohen manuellen Aufwand beim Implementieren der Tests hat. Das kostenpflichtige Framework „Quest Code Tester for Oracle" automatisiert die Erstellung von Tests zwar größtenteils, erfordert jedoch mehr Aufwand bei der Anbindung an einen Continuous-Integration-Prozess.

Als Build-Tool bietet sich das im Java-Umfeld mittlerweile weit verbreitete Open-Source-Produkt Maven an. Dieses zeichnet sich vor allem durch einen deklarativen Ansatz aus. Maven basiert auf einer Plugin-Architektur und sieht als Konvention einen Build-Lifecycle vor. Das Mapping einzelner Plugins auf die Phasen des Life-Cycle geschieht über Project-Object-Model (POM). Für uns soll Maven vor allem zwei Aufgaben erfüllen: Das Datenbank-Schema oder dessen Änderungen sollen in die Zielumgebung ausgeliefert, die Unit-Tests ausgeführt und deren Ergebnisse in einem für die Weiterverarbeitung geeigneten Format dargestellt werden.

Wie bereits erwähnt, ist die Steuerung der Auslieferung eine der größten Herausforderungen. Eine Empfehlung ist hier schwierig abzugeben, weil sie von vielen Faktoren abhängt. Für den einfachsten Fall steht ein SQL-Maven-Plugin (http://mojo.codehaus.org/sql-maven-plugin) zur Verfügung. Mit diesem lassen sich SQL-Befehle und -Skripte ausführen. Leider werden die SQL*Plus-Syntax sowie die Verschachtelung der Skripte nicht unterstützt. Existiert im Projekt bereits ein Auslieferungsverfahren basierend auf SQL-Skripten, würde es durchaus Sinn ergeben, mittels Exec-Maven-Plugins (http://mojo.codehaus.org/exec-maven-plugin) SQL*Plus aufzurufen. Wenn noch kein Verfahren zur Auslieferung festgelegt ist, kann man an das Anbinden eines sogenannten „Datenbank-Refactoring–Tools" denken. Es gibt hier auch einige Alternativen aus dem Open-Source-Bereich:

- Autopatch (http://autopatch.sourceforge.net)
- Liquibase (http://www.liquibase.org)
- dbdeploy (http://dbdeploy.com)
- MIGRATEDB (http://migratedb.sourceforge.net)

Um Unit-Tests auszuführen, steht ein Maven-utPLSQL-Plugin zur Verfügung (http://code.google.com/p/maven-utplsql-plugin). Dieses lässt die Unit-Tests in der Datenbank laufen und stellt die Ergebnisse in einer von der CI-Engine interpretierbaren Form dar (Surefire-Report). Gesteuert wird der ganze Prozess von einem CI-Server. Hier fällt die Wahl auf das Open-Source-Produkt Hudson (http://hudson-ci.org). Es bietet einen großen Funktionsumfang, ist

```
CREATE OR REPLACE PACKAGE order_util IS

FUNCTION is_order_shipped (in_order_id IN NUMBER) RETURN NUMBER;

END;
/

CREATE OR REPLACE PACKAGE BODY order_util IS

FUNCTION is_order_shipped (in_order_id IN NUMBER) RETURN NUMBER IS
l_status CHAR(15) := 'ausgeliefert';
l_res NUMBER;
BEGIN
   SELECT 1
     INTO l_res
     FROM orders
    WHERE order_id = in_order_id
      AND status = l_status;
   RETURN l_res;
EXCEPTION
   WHEN no_data_found THEN
     RETURN 0;
END is_order_shipped;
```

Listing 2: Datei order_util. sql

```
CREATE OR REPLACE PACKAGE ut_order_util IS

   PROCEDURE ut_setup;
   PROCEDURE ut_teardown;
   PROCEDURE ut_is_order_shipped;

END;
/

CREATE OR REPLACE PACKAGE BODY ut_order_util IS

PROCEDURE ut_setup IS
BEGIN
  INSERT INTO orders (order_id, order_date, status)
  VALUES (-999, sysdate, 'ausgeliefert');
END;

PROCEDURE ut_teardown IS
BEGIN
  DELETE FROM orders WHERE order_id = -999;
END;

PROCEDURE ut_is_order_shipped  IS
BEGIN
   utAssert.eq ('Order shipped'
               , order_util.is_order_shipped(-999)
               , 1
                );
END;

END;
/
```

Listing 3: Datei ut_order_util.sql

sehr gut mit Subversion und Maven integriert und in der Java-Community weit verbreitet.

Beispiel

Ein kleines Beispiel zeigt, wie das Zusammenspiel der oben beschriebenen Komponenten und Prozesse aussieht. Es besteht aus einer einzigen Tabelle „ORDERS“, in der Auftragsnummer, -datum und -status festgehalten sind. Der Datentyp der Spalte „STATUS“ ist als „CHAR(15)“ definiert. Entsprechende DDL-Befehle stehen im Skript „tab_orders.sql“ (siehe Listing 1).

```
DROP TABLE orders;
CREATE TABLE orders (order_id
NUMBER(10)
                   , order_date
DATE
                   , status
CHAR(15));
```

Listing 1: Datei tab_orders.sql

Das Package „ORDER_UTIL“ (siehe Listing 2) stellt die Funktion „is_order_shipped“ zur Verfügung, die für ausgelieferte Aufträge die Zahl „1“ zurückliefert und andernfalls „0“. Ein erfahrener PL/SQL-Entwickler wird schnell merken, dass die Implementierung nicht optimal ist. Die Variable „v_status“ hat den Typ „CHAR(15)“ statt der vorteilhafteren Deklaration „orders.status%TYPE“. Dies geschah bewusst so, um die Vorteile von Unit-Tests zu verdeutlichen.

Nun wird ein Package „ut_order_util“ erstellt, in dem sich die Test-Prozedur für die Funktion befindet sowie eine Prozedur „ut_setup“ mit benötigten Vorbereitungsschritten und „ut_teardown“ mit Aufräumarbeiten nach den Tests (siehe Listing 3). Zur Vorbereitung legt man einen Auftragsdatensatz mit dem Status „ausgeliefert“ an. Für diesen Datensatz wird nun die Funktion aufgerufen und das Ergebnis ausgewertet. Nach dem Test wird dieser Datensatz gelöscht.

Nun kommt die einfachste Möglichkeit zum Einsatz, um diese Datenstrukturen und Code in die Integrations-Datenbank auszuliefern – ohne Einsatz spezieller Tools: Ein „Installa-

tionsskript“ wird in SQL*Plus angelegt und aufgerufen (siehe Listing 4 und 5). Anzumerken ist, dass dieser Ansatz in realen Projekten sehr schnell an seine Grenzen stößt und ein „schlauerer“ Auslieferungsprozess benötigt wird.

```
connect ci/ci@test_db
@tab_orders.sql
@order_util.sql
@ut_order_util.sql
exit;
```

Listing 4: Datei install.sql

```
sqlplus -s /nolog @install.sql
```

Listing 5: Datei install.bat

Jetzt wird dem Build-Tool mitgeteilt, wie das Projekt aufzubauen ist. In Maven geschieht dies über „Project Object Model“ und die Datei „pom.xml“. Listing 6 zeigt einen Auszug. An die Lyfe-Cycle-Phase „compile“ ist die Ausführung des Exec-Plugins gebunden. Dabei wird das Skript „install.bat“ ausgeführt. Das zweite Plugin (maven-utplsql) ist an die Phase „process-test-resources“ angebunden. Diesem teilen wir noch die Connect-Informationen mit sowie, für welches Package wir die Unit-Tests ausführen wollen. Man kann auch die utPLSQL-Test-Suite ausführen lassen.

Nachdem alle Dateien in Subversion eingecheckt sind, legt man einen neuen Job in Hudson an. Dabei teilt man den Pfad zum VCS-Repository mit, wählt „Build Maven2 Project” aus und stellt ein, wann der Build initialisiert werden soll. Mit „Poll SCM“ ist die Konfiguration abgeschlossen. Nach einigen Sekunden startet der Build und Hudson informiert darüber, dass alles reibungslos funktioniert hat (siehe Abbildung 2).

Nach einiger Zeit entscheidet ein Entwickler, den Datentyp der Spalte „STATUS“ in der Tabelle „ORDERS“ von „CHAR(15)“ auf „VARCHAR2(15)“ zu ändern. Entsprechend legt er ein neues Skript „alter_tab_orders.sql“ an, passt „install.sql“ an und checkt beide Dateien in Subversion ein.

```
ALTER TABLE orders
MODIFY (status VARCHAR2(15));
```

Listing 7: Datei alter_tab_orders.sql

```
connect ci/ci@test_db
@alter_tab_orders.sql
exit;
```

Listing 8: Datei install.sql

Hudson erkennt diese Änderung und initiiert daraufhin einen neuen Build. Diesmal klappt es jedoch nicht mehr fehlerfrei (siehe Abbildung 3). Die Auslieferung hat noch funktioniert, aber der Unit-Test schlug fehl.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" ...      <modelV-
ersion>4.0.0</modelVersion>
       <groupId>com.trivadis.plsqltest</groupId>
       <artifactId>plsqltest</artifactId>
       <packaging>jar</packaging>
       <version>1.0-SNAPSHOT</version>
       <name>CI Example</name>
     ...
     <build>
        <plugins>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>exec-maven-plugin</artifactId>
        <version>1.1</version>
        <executions>
          <execution>
            <id>deploy</id>
            <phase>compile</phase>
            <goals>
              <goal>exec</goal>
            </goals>
       <configuration>
          <executable>install.bat</executable>
          <workingDirectory>${basedir}\src\main\</workingDirectory>
        </configuration>
          </execution>
        </executions>
      </plugin>
      <plugin>
        <groupId>com.theserverlabs.maven.utplsql</groupId>
        <artifactId>maven-utplsql-plugin</artifactId>
        <version>1.0-SNAPSHOT</version>
        ...
        <configuration>
          <driver>oracle.jdbc.driver.OracleDriver</driver>
          <url>jdbc:oracle:thin:@local:1521:test_db</url>
          <username>ci</username>
          <password>ci</password>
          <packageName>order_util</packageName>
        </configuration>
        <executions>
          <execution>
            <id>run-plsql-test-packages</id>
            <phase>process-test-resources</phase>
            <goals>
              <goal>execute</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Listing 6: Datei pom.xml

Abbildung 2: Erfolgreicher Build

Abbildung 3: Es sind Probleme aufgetreten

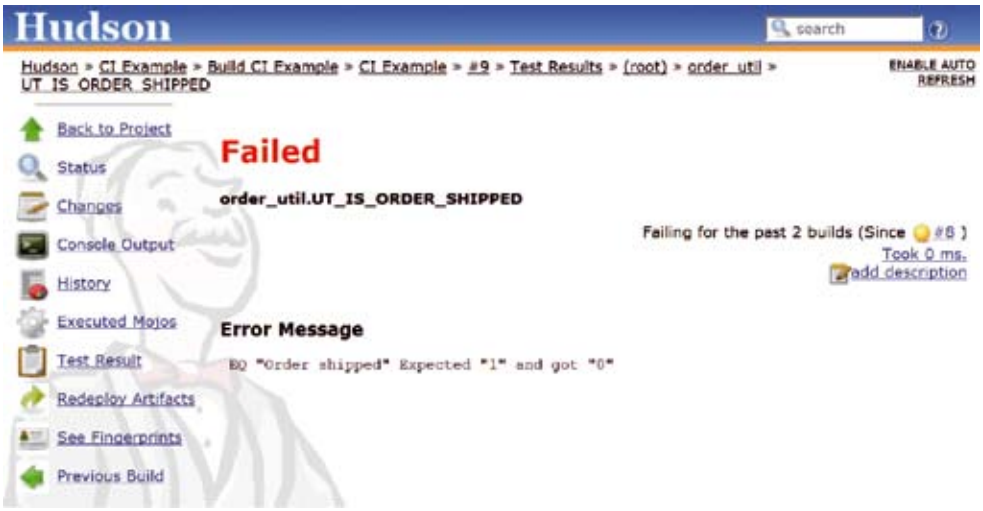

Abbildung 4: Unit-Test meldet einen Fehler

Abbildung 4 zeigt, welcher Test nicht erfolgreich war, sowie die entsprechende Meldung dazu. In der Tat: Wegen unglücklicher Variablen-Deklaration in der zu testenden Funktion „is_order_shipped“ funktioniert der Vergleich im SQL-Statement nicht mehr, weil die Variable vom Typ „CHAR“ noch abschließende Leerzeichen beinhaltet.

Dieses kleine Beispiel zeigt unter anderem, wie wichtig und nützlich selbst ganz triviale und offensichtliche Unit-Tests sein können. Man hätte normalerweise bis zum Zeitpunkt, an dem ein Fehlverhalten des Systems aus der Produktion gemeldet worden wäre, garantiert keinen Fehler bemerkt, da die Auslieferung funktioniert hat. Auch der PL/SQL-Code war gültig und kompilierbar. Nur das Verhalten der Funktion „is_order_shipped“ hat sich dramatisch verändert.

Fazit

Es ist durchaus möglich, einen Continuous-Integration-Prozess für Anwendungen in PL/SQL zu etablieren. Eine Infrastruktur aus Open-Source-Tools, die sich auf dem Markt zum De-facto-Standard vorgearbeitet haben, lässt sich aufbauen. Für diejenigen, die ihre Datenbank-Objekte bereits über ein Versionskontrollsystem verwalten und in der Lage sind, alle Änderungen daran vollautomatisch auszuliefern, wird der Aufwand hauptsächlich bei der Entwicklung der Unit-Tests liegen. Für alle anderen gilt es zunächst, diese Themen anzusprechen.

Andrej Pashchenko
Trivadis AG
andrej.pashchenko@trivadis.com