Continuous Database Integration

Andrej Pashchenko
Trivadis GmbH
Düsseldorf

Schlüsselworte
PL/SQL, Continuous Integration, Hudson, Subversion, Datenbank-Build, Datenbank-Change-Tool, Unit-Tests, Quest Code Tester for Oracle

Einleitung
Continuous Integration ist eine mittlerweile bekannte und verbreitete Best Practice. Die Idee ist einfach: Statt die Integration als nachgelagerte, schwer planbare Phase in einem Softwareentwicklungsprozess zu sehen, wird mit der Integration früh angefangen und es wird kontinuierlich integriert. Die Vorteile liegen auf der Hand: Integriert wird immer „stückchenweise" und somit wird 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 dabei schnell und agil auf sich ständig ändernde Fachanforderungen reagieren.

Während Continuous Integration beispielsweise im Bereich der Java-Entwicklung bereits alltäglich ist und eine Reihe von erfolgreich eingesetzten Tools vorweisen kann, ist der Bereich der datenbanknahen Software-Entwicklung, wo ein (Groß-)Teil der Geschäftslogik in PL/SQL implementiert ist, etwas zurückgeblieben. Woran liegt das? Wäre CI einfach anwendbar oder gibt es hier besondere Herausforderungen? Welche Komponenten fehlen einer typischen CI-Umgebung auf Basis von Hudson/Jenkins, Maven, Subversion & Co? Auf diese Fragen wird im Vortrag eingegangen und mögliche Lösungsansätze werden präsentiert.

Continuous Database Integration

Die wichtigste Voraussetzung bei der CI ist, dass man einen vollautomatischen Auslieferungsprozess (Build) hat, der auch Unit-Tests beinhaltet. Offensichtlich genügt es 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, müssen solche Änderungen mit ausgeliefert werden.

Genau hier liegt der größte Unterschied zum klassischen Prozess, wo Build-Artefakte jedes Mal aus dem Quellcode komplett neu aufgebaut werden. Beispielsweise kann man eine produktiv genutzte Tabelle nur durch ALTER-DDL-Befehle ändern. Auch an eine Datenmigration muss man im Normalfall denken.

Man kann argumentieren, dass es im Testsystem möglich ist, die ganzen Datenbank-Schemata immer neu anzulegen. Der Nachteil dieses Verfahrens ist es 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 vom CI-Prozess, wenn dadurch nicht nur die Funktionstüchtigkeit des PL/SQL-Codes nach jeder Änderung bestätigen ließe, sondern auch der Auslieferungsprozess an sich immer mit getestet würde.

Es geht vielmehr darum, ein existierendes Datenbank-Schema kontinuierlich zu verändern und die Änderungen kontinuierlich zu integrieren.

Build-Prozess

Bei dem Build-Prozess geht es zunächst darum, Änderungen, die als DDL-Skripte im VCS vorliegen, an die Zieldatenbank auszuliefern (siehe Abb. 1). Dafür ist ein sogenanntes DB-Change-Tool zuständig.

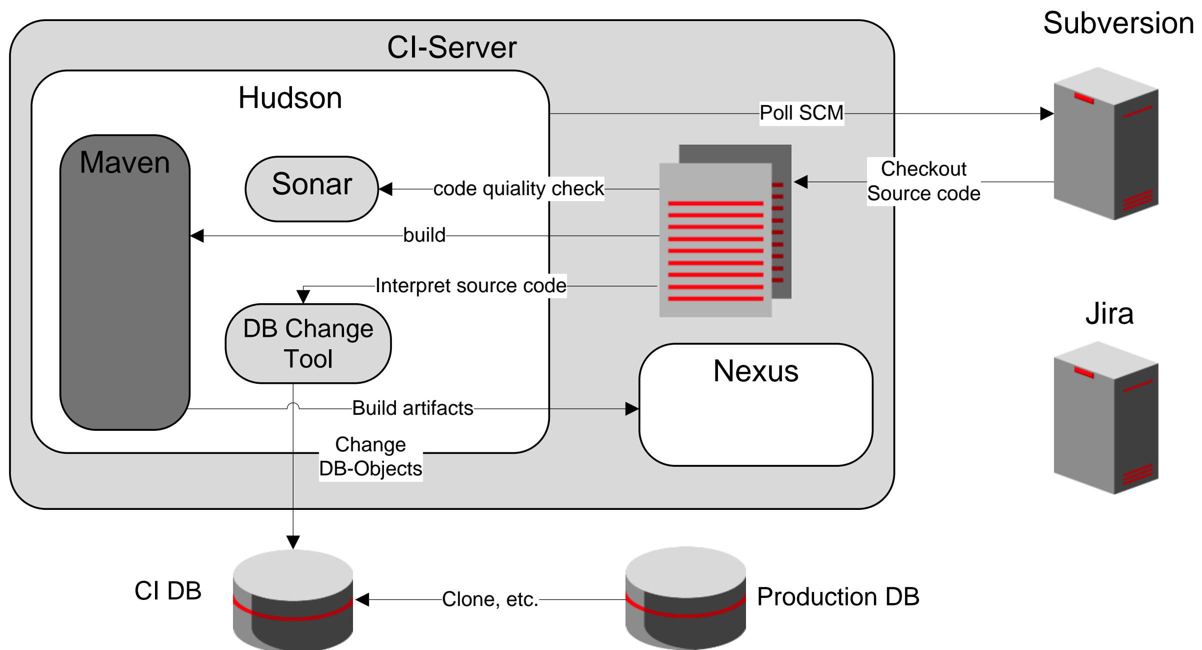

Abb. 1: CI-Umgebung

Dieses Tool hilft, die Auslieferungslogik aus dem Quelltext auszulagern und verwaltet z.B. folgende Aspekte: Was ist bereits ausgeliefert worden? Was ist in einer Fehlersituation zu tun? In welcher Reihenfolge sollen die Änderungen angewendet werden? Wie wird das Ausführen protokolliert? Diese Aufgaben sind sehr wahrscheinlich in jedem Projekt gleich oder ähnlich. Daher macht es Sinn, sie an ein Tool zu delegieren.

Es gibt eine Reihe von derartigen Tools auf dem Open-Source-Markt. Leider haben sie alle gemeinsame Nachteile: sehr dünne Entwickler und Benutzerbasis, sparsame Dokumentation, Unsicherheit wegen Weiterentwicklung.

Eine Eigenentwicklung wäre aus diesen Gründen durchaus in Betracht zu ziehen. Dazu kommt, dass man eine bessere Abstimmung zwischen dem Auslieferungsprozess und der Art und Weise wie Quellcode organisiert ist (Verzeichnisstruktur, Inhalt der Dateien, Verbindungsaufbau). Auch die Möglichkeit, in diesem Fall Oracle Client und SQL*Plus anstatt JDBC zu verwenden, bringt an der Stelle Vorteile.

Um die erwähnten Anforderungen zu erfüllen, wird eine Reihe von Shell und SQL-Skripten verwendet. Die Vorgehensweise beim Ausliefern ist auf der Abbildung 2 schematisch dargestellt.

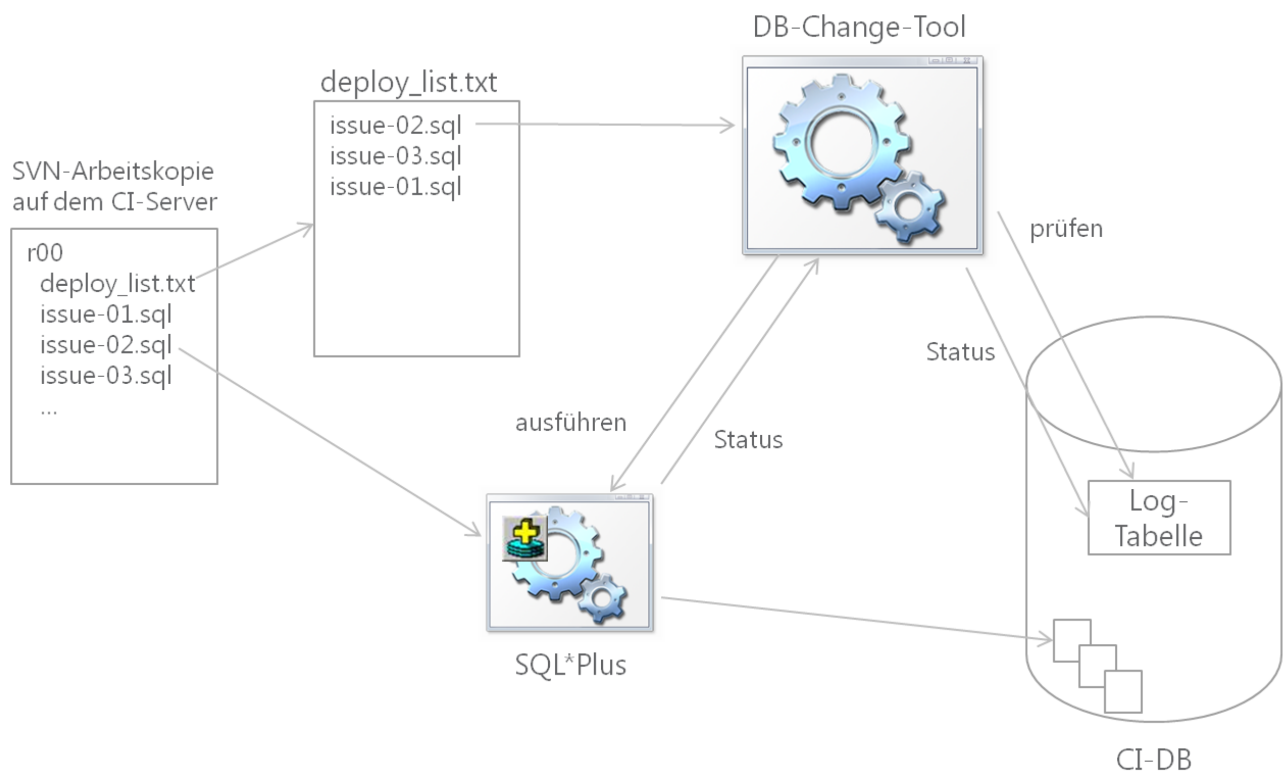

Abb. 2: Funktionsweise DB-Change-Tool

Die Anbindung in den Build-Prozess, der von Maven gesteuert wird, erfolgt mithilfe vom exec-maven-plugin.

Eine wichtige Frage, wenn es um automatisiertes Ausführen von SQL-Skripten geht, wo und wann erfolgt der Verbindungsaufbau und wo werden Zugangsdaten gespeichert. Eine Hilfe schafft hierbei das Client-Feature Oracle Secure External Password Store. Das Konzept hierbei ist, dass die Verbindungsdaten verschlüsselt in einem Wallet abgespeichert werden.

Nach der einfachen Konfiguration besteht die Möglichkeit, sich ohne das Passwort zu verbinden, z.B.

```
connect /@db_tns_alias
```

Die Auflösung erfolgt über den TNS-Eintrag. In diesem Fall ist es leider nicht möglich, den Datenbankbenutzernamen zu übergeben. Die Folge: man braucht für jedes Schema, mit dem man eine Verbindung aufbauen möchte, einen eigenen TNS-Eintrag. Dieser einmalige Mehraufwand bei der Konfiguration zahlt sich aber durchaus aus: Die Änderungsskripte werden übersichtlicher und können thematisch zusammengestellt werden, auch wenn die Änderungen über mehrere Datenbankschemata erfolgen. So kann beispielsweise ein Skript aussehen, das die Datenstrukturen anpasst, um eine neue Spalte in einem Datawarehouse in mehreren ETL-Schichten einzuführen:

```
-- View-Definition in der Quell-DB
connect /@crm_at_db_src
@../crm_at_db_src/views/kunde.vw

-- Tabelle im Stage erweitern
connect /@stage_at_db_dwh

ALTER TABLE stg_kunde ADD (cust_category VARCHAR2(100));

-- Tabelle im Core erweitern
connect /@core_at_db_dwh

ALTER TABLE co_kunde ADD (cust_category VARCHAR2(100));

-- Dimension im Mart erweitern
connect /@mart_at_db_dwh

ALTER TABLE dim_kunde ADD (cust_category VARCHAR2(100));
```

Tests für PL/SQL

Für die Tests von PL/SQL-Code gibt es schon seit langem einige Open-Source-Lösungen, wie utplsql, Pluto, etc. Wenn man aber mehr Wert auf das Komfort und Flexibilität legt, bietet sich ein kommerzielles Produkt an: Quest Code Tester for Oracle (QCTO).

In der letzten Version haben die Entwickler viel getan, um die Automatisierung und die Integration in eine CI-Umgebung zu vereinfachen. Das Framework bietet eine PL/SQL API, über die sich viele Aufgaben automatisieren lassen. Um das Tool an eine CI-Umgebung anzubinden, soll man vor allem folgende drei Themen adressieren: die Testdefinitionen sollen über ein Versionskontrollesystem verteilt werden, sie sollen automatisch ausführbar sein und die Testergebnisse sollen von der CI-Engine interpretiert werden können.

Das Verteilen der Tests kann man über die Export/Import-Möglichkeit im XML-Format lösen.

```
begin
 -- Load the export, preparing for import.
  qu_xmldom_import.init_xml(:xml_import);
  qu_xmldom_import.set_options( … );
  qu_xmldom_import.set_for_mapping(prog_owner_in=>'HR'
                                  ,harn_owner_in=>'HR');
  -- Perform the import.
  qu_xmldom_import.import_as_xml;
end;
/
```

Die Herausforderung hierbei ist es, dass die Import-Prozedur als Parameter eine CLOB-Variable mit dem Inhalt der XML-Export-Datei braucht. Kann man aus der Datenbank auf die Datei zugreifen, vereinfacht sich die Aufgabe. Sonst muss man die Datei vom Client (CI-Server) hochladen.

Für die Ausführung von einzelnen Tests oder Testsuites bietet QCTO auch eine PL/SQL API: das Package QU_TEST mit verschiedensten Varianten, was die Ein- und Ausgabeparameter angeht. Das Package ist gut dokumentiert.

Wenn die Tests auf diese Weise ausgeführt worden sind, ist Hudson noch nichts davon bekannt. Diese Lücke kann man aber relativ einfach schließen. Maven ruft für die Build-Phase „test" standardmäßig surefire-plugin auf, das JUnit-Tests ausführt und deren Ergebnisse in JUnit-XML-Format aufzeichnet. Dieses Format ist recht einfach. Es bietet sich an, die Rückgabeparameter von den Prozeduren aus dem Package QU_TEST zu parsen oder alternativ direkt auf die Tabellen vom Code Tester zuzugreifen und die Ergebnisse der Ausführung in diesem XML-Format als Datei im Standardverzeichnis *target/surefire-reports* abzulegen. Über diese Dateien ist Hudson in der Lage, die Informationen über die ausgeführten Unit-Tests anzuzeigen und sie auszuwerten.

Fazit

Mit einfachen technischen Mitteln lassen sich zwei wichtigsten Themen aus dem Bereich Continuous Database Integration adressieren. Die Logik ließ sich alleine mit Shell- und SQL-Skripten abbilden. Im Rahmen von Maven-Build-Prozess werden diese über Exec-Plugin aufgerufen.

Kontaktadresse:
Andrej Pashchenko
Trivadis GmbH
Werdener Str. 4
D-40227 Düsseldorf

Telefon:	+49 (0) 211-58 66 64 70
Fax:	+49 (0) 211-58 66 64 71
E-Mail	andrej.pashchenko@trivadis.com
Internet:	www.trivadis.com