```
cd /opt/oracle/oak/onecmd
./GridInst.pl -r 1-11
```

Listing 6

```
chown grid:asmadmin /dev/mapper/HDD*p*
chown grid:asmadmin /dev/mapper/SSD*p*
chmod 660 /dev/mapper/HDD*p*
chmod 660 /dev/mapper/SSD*p*
```

Listing 7

```
cd $ORACLE_HOME/addnode ./addnode.sh -silent CLUSTER_NEW_
NODES={odanode1} CLUSTER_NEW_VIRTUAL_HOSTNAMES={odanode1-vip}
```

Listing 8

```
[oracle@odanode2 ~]$ cd /u01/app/oracle/product/12.1.0.2/dbhome_1/
addnode/ [oracle@odanode2 addnode]$ ./addnode.sh -silent CLUSTER_NEW_
NODES={odanode1}
```

Listing 9

keiten und Installationsvorgänge, die auf der ODA automatisiert ablaufen.

Die ODA ist noch nicht reif genug, um mit einem Klick alles wiederherzustellen, der menschliche Faktor spielt daher bei Wartung und Betrieb immer noch eine sehr große Rolle. Ein gesundes Basiswissen zu dem, was während des ODA-Deployments unter der Haube passiert, ist also nicht zu unterschätzen und verschafft uns einen anderen Blickwinkel auf die bunte ODA-Welt mit Tools wie dem App-Manager.

Quellen

[1] NOTE 1352884.1: „opatch auto“ or „rootcrs.pl -unlock“ or „roothas.pl -unlock“ or „addNode.sh“, Hangs Due to Many Audit Files in „GRID_HOME“
[2] NOTE 1526405.1: „addNode.sh“-Fail PRCF-2023, The following contents are not transferred as they are non-readable
[3] NOTE 2027830.1: ODA HA (High Availability) Deployment Step Descriptions, A List of Deployment Version Specific Steps Used for Each ODA HA Version Using GridInst.pl
[4] NOTE 1373599.1: ODA Oracle Database Appliance Bare Metal Restore Procedure X5-2 and X6-2
[5] NOTE 888888.1: Oracle Database Appliance, 18.1, 12.X, and 2.X Supported ODA Versions & Known Issues (Doc ID 888888.1)

Andrzej Rydzanicz
andrzej.rydzanicz@opitz-consulting.com

Continuous Integration in der Datenbank-Entwicklung

Dominic Weiser, ISTEC Industrielle Software-Technik GmbH

Betrachtet man die aktuellen Arbeitsweisen in der Datenbank-Entwicklung, stellt man sehr schnell fest, dass die agile Vorgehensweise schon ihren Einzug gehalten hat. Ständig wechselnde und sich an die Aufgaben anpassende Entwicklungsteams sind eine Ausprägung dieser Vorgehensweise.

In den seltensten Fällen sind Projekte und Aufgaben so simpel, dass sie von einem Entwickler zu jeder Zeit vollumfänglich überschaut werden können. An dieser Stelle hilft Continuous Integration dabei, negative Nebeneffekte auf fremdem oder eigenem Code frühzeitig aufzuspüren und zu beseitigen. Dabei ist Continuous Integration bestimmt kein Novum mehr, jedoch hat es den flächendeckenden Einzug in die Datenbank-Entwicklung noch nicht wirklich geschafft, obwohl bereits alle technologischen Hilfsmittel in Fülle am Markt verfügbar sind. Dieser Artikel beschreibt die Grundlagen von Continuous Integration und zeigt eine bei-

spielhafte Implementierung für den eigenen Entwicklungs-Workflow.

Wer sich mit Continuous Integration beschäftigt, wird sehr schnell auf unterschiedliche Definitionen und Herangehensweisen treffen. Um das Thema von Grund auf zu betrachten, befasst man sich zunächst mit der Definition nach Martin Fowler, wie er sie im April 2006 mit der aufkommenden Popularität von Extreme-Programming aufgestellt hat: „Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly" *[1]*.

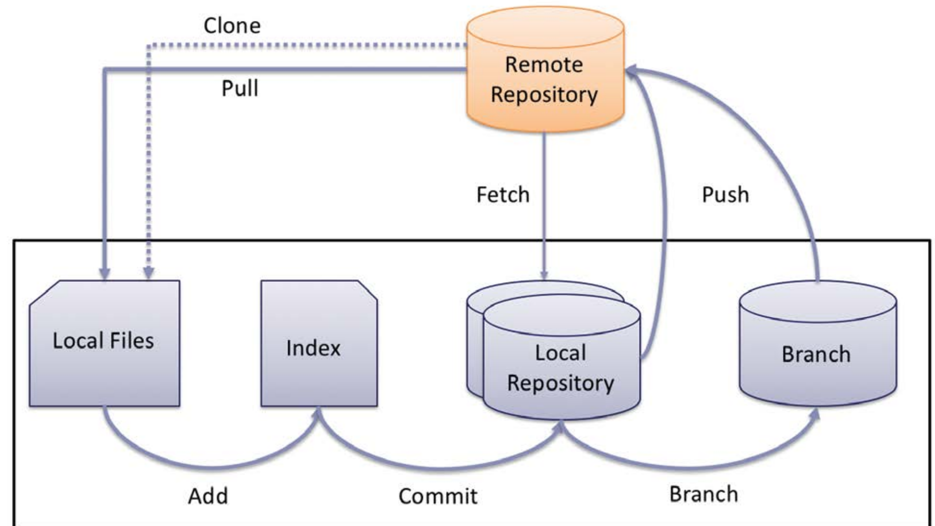

Abbildung 1: Git-Übersicht

Technische Anforderungen

Aus dieser Definition von Continuous Integration lassen sich sehr einfach die genauen technologischen Anforderungen für die Realisierung ableiten:

- *Source-Verwaltungssystem („...each person integrates at least daily... ")*
  Jeder Entwickler hat seinen Entwicklungsstand mindestens täglich in das Verwaltungssystem zurückzuspielen. Zwar könnte hiermit auch das einfache Ablegen auf einem gemeinsamen Verzeichnis gemeint sein, jedoch bieten uns moderne Source-Verwaltungssysteme echte Mehrwerte wie Änderungsverfolgung oder Branching.

- *Build-System („...verified by an automated build... ")*
  Der zurückgespielte Code wird auf einer neutralen Umgebung automatisiert gebaut.

- *Automatisierte Tests („...including test ...")*
  Der zurückgespielte und gebaute Code muss anschließend automatisiert getestet werden.

Source-Verwaltung: Git

Im Allgemeinen dienen Source-Verwaltungssysteme dazu, Änderungen an Dateien zu erfassen und für den Anwender nachvollziehbar zu machen. Wenn man an dieser Stelle in den Markt hört, gibt es unter Software-Entwicklern nur noch eine Wahl: Git *[2]*. Jeder spricht davon und jeder setzt es ein. Wirklich alle? Nein, denn es gibt noch andere weitverbreitete Source-Verwaltungssysteme wie CVS, Subversion (SVN) oder Mercury. Jedoch für welches System soll man sich entscheiden oder sollte das alte System vielleicht migriert werden?

Um diese Frage zu beantworten, muss man sich die Frage stellen, ob man eine zentrale oder eine verteilte Verwaltung anstrebt. Denn genau darin liegen die gravierenden Unterschiede. Bei CVS oder SVN liegen die zurückgespielten Dateien an einem Ort. Dies kann sehr hilfreich sein, wenn der entwickelte Code in genau einem Projekt oder Produkt zum Einsatz kommt. Alternativ dazu verwendet Git einen verteilten Ansatz und ist darauf ausgelegt, einfache Abspaltungen und Modularisierungen zu erzielen.

So ist es möglich, einzelne Programmteile aus unterschiedlichen Repositories lokal zusammenzuführen und weiterzuentwickeln. Anschließend lassen sich die lokalen Änderungen wieder in die verteilten Repositories zurückspielen. Git ist besonders stark darin, Konflikte in den Code-Versionen aufzuspüren und selbstständig zu beheben. Somit ist der Umgang mit Branches um einiges einfacher und handlicher *(siehe Abbildung 1)*.

Zu Beginn der Arbeit in einem neuen Projektteam muss der bestehende Source-Code heruntergeladen werden. Dazu wird der Befehl „git clone /pfad/zum/repository" ausgeführt. Er führt gleich zwei Aktionen durch; erstens werden die so geklonten Dateien im lokalen Dateisystem abgelegt und zweitens wird ein lokales Repository erstellt.

Das Zurückspielen der Änderungen des lokalen Repository ist die häufigste Fehlerquelle bei Umsteigern von CVS/SVN. Die benötigten Befehle lauten „git add <dateiname>" und „git commit -m „Commit-Nachricht"". Mit „add" werden die Änderungen dem lokalen Index hinzugefügt; ein sehr gutes Mittel, sich Zwischenstände für einen späteren Commit zu merken.

Gerade wenn man an großen Änderungen arbeitet, kann es sonst recht schnell dazu kommen, dass eine Source beim Zurückspielen übersehen wird. Commit schreibt die Änderungen in das lokale Repository. Hier liegt die Fehlerquelle: Dieser Code ist noch nicht für die Kollegen verfügbar. Mit „git push origin master" müssen die Änderungen vom lokalen Repository (origin) erst in das remote Repository (master) übertragen werden. Dies wäre der direkteste Weg, die eigenen Änderungen zu teilen. Ein Blick in die vielen Open-Source-Projekte zeigt, dass Änderungen in der Regel als eigener Branch zurückgespielt werden. So sorgt der Push nur dafür, dass eine Anfrage zum Zurückspielen gestellt wird. Ein Verwalter kann die Änderungen so einfacher nachvollziehen und den Push-Request gegebenenfalls sogar ablehnen.

Build-System: Jenkins – Maven

Beim Blick ins Java-Umfeld stellen wir fest, dass Build-Systeme seit ca. 2008 ein fes-

ter Bestandteil des Entwicklungsprozesses sind. Der prominenteste Vertreter ist Jenkins *[3]*. Es handelt sich hierbei um einen Fork des Oracle-Produkts Hudson, dessen Weiterentwicklung zugunsten von Jenkins eingestellt wurde. Build-Systeme dienen dazu, den entwickelten Code auf einer neutralen Umgebung zu kompilieren. Somit lässt sich die Integrität des Codes unabhängig von der eigenen Entwicklungsumgebung gewährleisten. Dazu wird die einzelne Kompilierungsausführung in sogenannten Jobs definiert. Ein Job beschreibt immer die Ausführung einer Ausführungsroutine, die sich durch die modulare Struktur von Jenkins durch eine Vielzahl von Plug-ins erweitern lässt. Für jeden Job gibt es unterschiedliche Einstiegspunkte *(siehe Abbildung 2)*.

Die häufigste Wahl fällt dabei auf die Commit- beziehungsweise Zeit-gesteuerten Ausführungen. Die Commit-gesteuerte Ausführung eignet sich, um die Kompilierbarkeit des Codes zu überprüfen; Zeit-gesteuerte Ausführungen werden immer dann verwendet, wenn ganze Testpläne oder Deployments gefahren werden.

Zwar gibt es für Jenkins bereits einige Plug-ins, die in der Lage sind, PL/SQL-Code zu kompilieren, jedoch gibt es da noch weitere Alternativen. Eine Alternative ist Maven *[4] (siehe Abbildung 3)*.

Maven kommt ebenfalls aus dem Java-Umfeld und sollte eigentlich dazu dienen, abhängige Bibliotheken einheitlich in den Code zu laden. Daraus entwickelte sich mit der Zeit ein modulares System, das um etliche Plug-ins erweitert wurde. So gibt es verschiedene Plug-ins, mit denen sich PL/SQL-Code kompilieren lässt. Wer zusätzlich neben dem Datenbank-Code auch ein Frontend oder Middleware mit zu verwalten hat, bekommt mit Maven eine All-in-one-Lösung.

Zum Bauen von SQL mittels Maven eignet sich am besten das native Command Line Interface (CLI) der Datenbank. Daher kommt in *Listing 1* das CLI von Oracle zum Einsatz. Die wichtigsten Informationen sind: der Pfad zum SQL*Plus, der Pfad zur Datenbank und zu den Quelldateien inklusive Build- und Testausführung. Im Beispiel wird nur ein Ausführungsziel („goal") einer Ausführungsphase des Maven-Builds dargestellt. In der Praxis bietet es sich an, die unterschiedlichen Ausführungen (Builds und Tests) in unterschiedlichen Zielen beziehungsweise Phasen ausführen zu lassen. Für sehr große Projekte kann die Trennung auch über Maven-Module durchgeführt werden.

Zurück zum Jenkins-Job: Dieser hat nun das Kompilieren über Maven und die Testausführung angestoßen. Die Ergebnisse sollen nun genauso automatisiert in den Entwicklungszyklus einfließen wie deren Ausführung. Dazu lässt sich Jenkins an moderne Kollaborationstools wie Slack oder Trello anbinden. Wer es dagegen klassisch haben möchte, lässt die verantwortlichen Entwickler per E-Mail benachrichtigen.

Automatisierte Tests: utPLSQL

Im Zusammenhang mit Software-Tests gibt es viele unterschiedliche Herangehensweisen. Ein nativer Ansatz wäre, jede Methode einzeln zu testen. Dieser Versuch würde jedoch sehr schnell dazu führen, dass jede geschriebene Methode durch enorm viel mehr an Test-Methoden ergänzt werden muss. Das würde den Entwicklungsaufwand nur unnötig in die Höhe treiben. Aus diesem Grund haben sich die funktionalen Tests (Unit Tests) durchgesetzt. Es müssen also nur noch die Funktionalitäten getestet werden. Dies kann auch dazu führen, dass ein Test sich auf mehrere Schichten (Frontend, Middleware und Backend) auswirkt.

Betrachtet man nun eine Möglichkeit, Unit-Tests auf der Datenbank auszuführen, wird man sehr schnell auf das utPLSQL-Projekt *[5]* aufmerksam. Es wurde ursprünglich durch Steven Feuerstein entwickelt. Seit dem Jahr 2017 wird es durch ein Team auf GitHub als Open-Source-Projekt aktiv weiterentwickelt.

Auch bei Unit-Test gibt es unterschiedliche Ausführungszeitpunkte. Per se setzt sich ein Unit-Test aus mehreren Test-Methoden zusammen. Jede davon lässt sich durch eine „Before"- und eine „After"-Methode ergänzen, die immer davor und danach ausgeführt werden. Diese Ausführungen lassen sich noch weiter verfeinern, wie zum Beispiel vor jeder Methode, vor der Klasse oder vor dem Package. So lässt sich eine benötigte Ausgangssituation vorbereiten beziehungsweise wiederherstellen. Zum Installieren von utPLSQL bietet dieses eine vorgefertigte Installationsdatei (.sql), die zwingend als SYSDBA ausgeführt werden muss: „sqlplus sys/sys_pass@db as sysdba @@install_headless.sql".

Listing 2 zeigt, wie mithilfe von utPLSQL ein Testpaket erzeugt und gestartet wird. In der Testprozedur wird die zu testende Funktion „btwnstr" aufgerufen. Diese soll als Ergebnis den Substring zwischen einem Start- und einem Endpunkt zurückgeben. Ist dies nicht der Fall, schlägt der Test fehl.

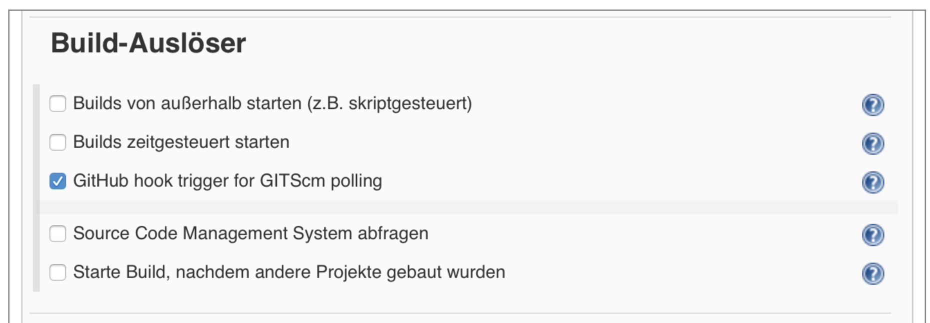

Abbildung 2: OnCommit-Job

Abbildung 3: Maven-Ziel

```
<plugin>
 <groupId>org.codehaus.mojo</groupId>
 <artifactId>exec-maven-plugin</artifactId>
 <version>${maven_exec_plugin_version}</version>
 <executions>
   <execution>
   <phase>run-test</phase>
   <goals>
       <goal>test</goal>
   </goals>
   <configuration>
       <executable>pfadzuOracle/bin/sqlplus</executable>
       <workingDirectory>/pfad/zum/repository</workingDirectory>
       <environmentVariables>
           <ORACLE_HOME>PFADZUORACLE</ORACLE_HOME>
           <PATH>$PATH:$ORACLE_HOME/bin:$JAVA_HOME/bin</PATH>
       </environmentVariables>
       <arguments>
           <argument>user/pwd@db_tns</argument>
           <argument>doTest.sql</argument>
       </arguments>
   </configuration>
   </execution>
 </executions>
</plugin>
```

Listing 1: Maven-Konfiguration

```
create or replace package test_betwnstr as
  -- %suite(Between string function)
  -- %test(Returns substring from start position to end position)
  procedure basic_firstTest;
end;
create or replace package body test_betwnstr as
  procedure basic_firstTest is
  begin
      ut.expect( betwnstr( '1234567', 2, 5 ) ).to_equal('2345');
  end;
end;
begin
      ut.run('test_betwnstr');
end;
```

Listing 2: „doTest.sql"

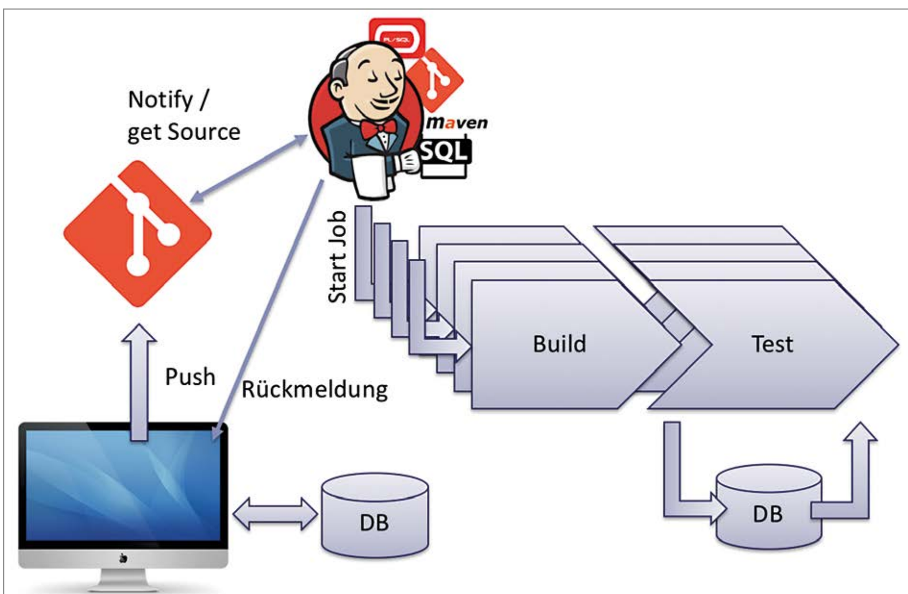

Abbildung 4: CI-Infrastruktur

Fazit

Werden die unterschiedlichen Technologien so wie in diesem Artikel beschrieben aufgesetzt, erhält man die in *Abbildung 4* gezeigte Infrastruktur. Der Entwickler behält seine lokale Entwicklungsumgebung und sollte mit dieser autark arbeiten. Nachdem die Änderungen umgesetzt wurden, werden diese mit Git in das globale Repository zurückgespielt und mithilfe von Jenkins automatisiert gebaut und getestet. Der Entwickler erhält zeitnah Informationen über seine Änderungen und kann gegebenenfalls verbessernd eingreifen.

Die im Artikel dargestellte Vorgehensweise zeigt eine stark vereinfachte Anwendung von Continuous Integration. Durch den modularen Aufbau der Technologien und durch die möglichen Plug-ins lässt sich diese Umgebung einfach an die eigenen Bedürfnisse anpassen.

Für das Mehr an Wartungs- und Entwicklungsaufwand erhält man eine Reduzierung der Rückmeldezeiten und weniger unvorhergesehene Fehler bei der produktiven Installation. Der Autor ist überzeugt, dass das Thema „Continuous Integration" mit seinen Ausprägungen „Continuous Deployment" und „Continuous Delivery" in Zukunft eine immer größere Verbreitung in der Datenbank-Entwicklung finden wird.

Weitere Informationen

[1] *https://www.martinfowler.com/articles/continuousIntegration.html*
[2] *https://git-scm.com*
[3] *https://jenkins.io*
[4] *https://maven.apache.org*
[5] *http://utplsql.org*

Dominic Weiser
dominic.weiser@istec.de