Continous Integration in APEX Projekten

Sven Böttcher

Apps Associates GmbH

Dortmund

Schlüsselworte
Application Express, Continuous Integration, Jenkins, Maven, Rspec, Selenium

Einleitung
Continous Integration (CI) ist ein etablierter Prozess, welcher in der Softwareentwicklung zur Verbesserung der Softwarequalität eingesetzt wird. Der Prozess besteht aus den grundlegenden und inhärenten Teilschritten der Versionierung des Quellcodes in einem Versionskontrollsystem (commit), dem ständigen Zusammenfügen der einzelnen Softwarekomponenten (build) sowie dem Ausführen von verschiedenartigen Softwaretests (test). Dieser Prozess lässt sich dabei durch einen CI Server, wie zum Beispiel Jenkins, weitestgehend automatisieren. Zudem gibt der CI Server den Projektverantwortlichen und den Entwicklern Rückmeldung über den aktuellen Zustand des Softwareprojekts, sodass auf nicht erfolgreiche Builds oder Tests schnellstmöglich reagiert werden kann.
Während Continous Integration beispielsweise in der Java Entwicklung schon seit langem angewendet wird, ist dieser Begriff in PLSQL bzw. APEX Projekten erfahrungsgemäß relativ unbekannt und findet entsprechend kaum Anwendung. Dies liegt unter anderem daran, dass sich die APEX Entwicklung grundlegend von der Programmierung in einer „konventionellen“ Programmiersprache unterscheidet und nicht direkt ersichtlich ist, wie sich der Continous Integration Prozess in APEX Projekten einsetzen lässt oder aber, dass der Continous Integration Prozess als sehr Aufwändig erscheint.

Klassische Continous Integration Architektur

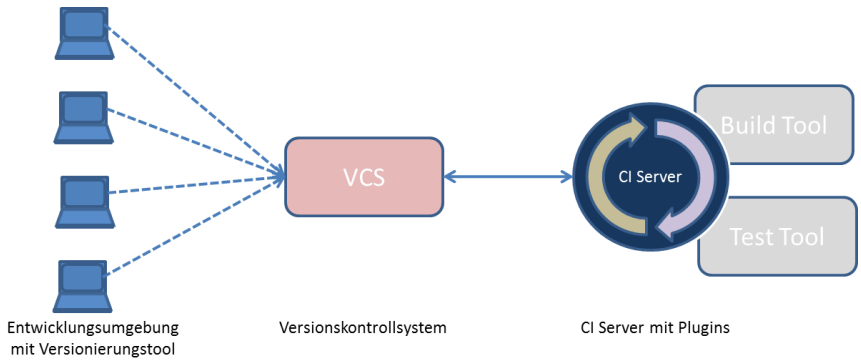

Abb. 1: Continous Integration Komponenten

Abbildung 1 stellt die klassischen Countinuous Integration Komponenten dar, wie sie in vielen Softwareprojekten eingesetzt werden. Diese bestehen aus
der Entwicklungsumgebung mit einem Versionierungstool,
einem Versionskontrollsystem und
einem CI Server mit diversen Plugins, z.B. zum automatisierten Bauen und Testen der Software.

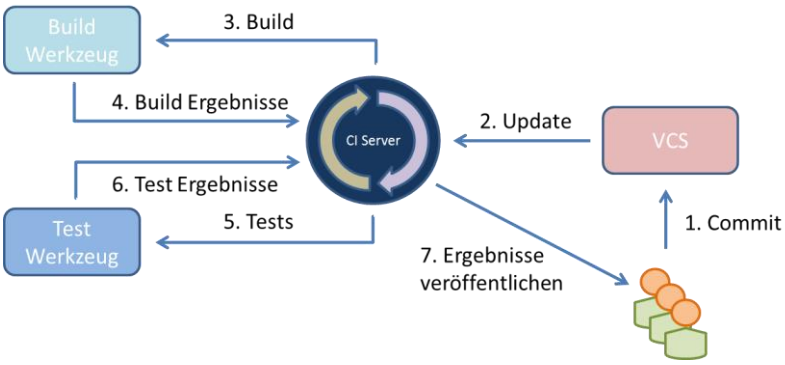

Abb. 2: Der Continous Integration Prozess

Der CI Prozess ist in Abbildung 2 dargestellt. Alle am Projekt beteiligten Entwickler aktualisieren in definierten und nicht zu langen Intervallen den von Ihnen entwickelten Quellcode im Versionskontrollsystem. Der CI Server prüft dieses Versionskontrollsystem in ebenso definierten (aber ggf. anderen) Intervallen auf Änderungen (polling) und übernimmt diese in ein lokales Repository. Nach jeder Änderung in der Codebasis oder in Abhängigkeit von Zeitplänen wird das Softwareprojekt kompiliert bzw. „gebaut". Nachdem dieser Schritt (erfolgreich) durchgeführt wurde, können automatisierte Testläufe durch den CI Server angestoßen werden. Um den Projektleitern und Entwicklern möglichst schnell ein Feedback über den aktuellen Stand des Projektes hinsichtlich von Programmierfehlern zu geben, verfügen CI Server i.A. über einen Feedback-Mechanismus in Form von HTML Seiten, auf denen aktuelle und historische Fehler aufgelistet werden.

Durch diesen Prozess entsteht der Vorteil, dass die gesamte Codebasis von allen beteiligten Entwicklern ständig integriert wird. So lassen sich insbesondere Softwaretests durchführen, die das Zusammenspiel von voneinander abhängigen Softwaremodulen testen, wodurch entsprechende Probleme bereits sehr früh erkannt und beseitigt werden können. Fallen solche Probleme erst in einem späten Entwicklungsstadium auf, kann eine Beseitigung sehr zeitaufwändig bzw. kostenintensiv werden oder sogar zum Scheitern des Projektes führen.

Continous Integration Architektur in APEX Projekten

In konventionellen Java-Entwicklungsprojekten verfügt in der Regel jeder Entwickler über eine lokale Arbeitskopie des Quellcodes und die Entwicklung erfolgt entsprechend lokal. Wird zum Beispiel Subversion (SVN) als Versionskontrollsystem eingesetzt, werden alle Änderungen in ein zentrales Repository übertragen. Erfahrungsgemäß entwickeln in PLSQL bzw. APEX Projekten zumeist alle Entwickler in einer zentralen Datenbank bzw. in einer zentralen APEX Instanz (vgl. Abbildung 3).

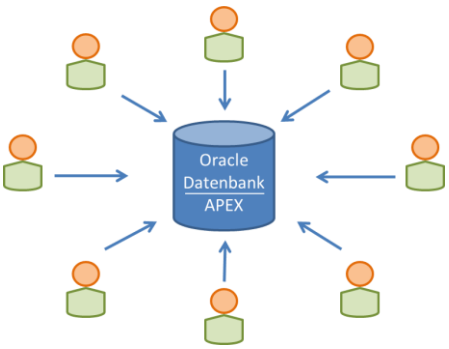

Abb. 3: Typische APEX Entwicklung im Team

Dadurch ergibt sich insbesondere das Problem, dass es bei nicht ausreichender Abstimmung zum Überschreiben von Quellcode oder APEX-Entwicklungen kommen kann. Dies ist insbesondere dann problematisch, wenn keine Versionierung eingesetzt wird oder Änderungen vor dem Übertragen in das Versionskontrollsystem überschrieben werden. Einen Ansatz dieses Problem zu umgehen und einen Continous Integration Prozess, ähnlich wie in JAVA-Projekten zu ermöglichen, ist in Abbildung 4 dargestellt.

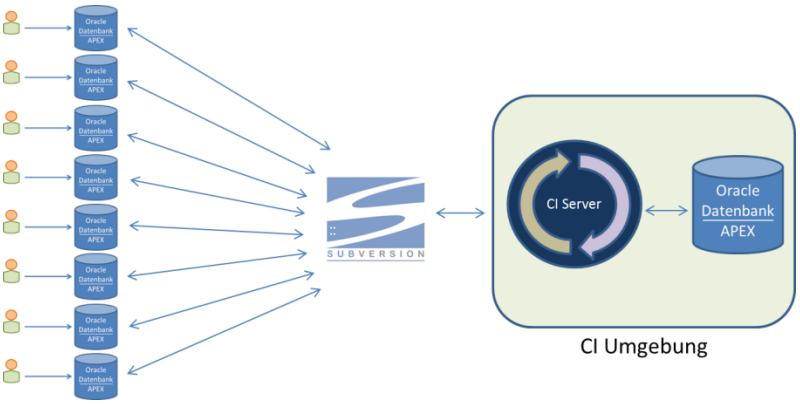

Abb. 4: Continous Integration in APEX Projekten

In diesem Szenario verfügt jeder Entwickler über eine lokale Datenbank und eine lokale APEX Instanz. Die benötigte Infrastruktur kann beispielsweise einfach in einer virtuellen Box installiert und an alle Entwickler verteilt werden. Die eigentliche Entwicklung erfolgt nun lokal und nicht mehr in einer zentralen Datenbank. Da Versionskontrollsysteme i.A. dateibasiert sind, muss der PLSQL Code sowie alle DDL/DCL Befehle vor dem Übertragen in das Versionskontrollsystem (z.B. SVN) in einer Datei gespeichert werden. Der CI Server prüft in definierten Intervallen das Versionskontrollsystem und aktualisiert seine eigene Codebasis. Bei Änderungen in der SQL-/PLSQL-/APEX-Codebasis werden diese in der CI Datenbank nachgezogen und automatisierte Tests durchgeführt.
Als CI Server kann Jenkins[1] in APEX Projekten sehr gut eingesetzt werden. Da Jenkins unter der MIT Lizenz steht, fallen für die Verwendung keine Lizenzkosten an. Zudem lässt sich Jenkins durch zahlreiche Plugins sehr gut an die projektspezifischen Anforderungen anpassen. Für APEX Projekte bedeutet dies, dass ein Build-Werkzeug zum Erzeugen der Datenbankobjekte und der APEX Anwendung sowie ein Test-Werkzeug zum Testen des PLSQL-Code und der APEX-Anwendung benötigt wird.

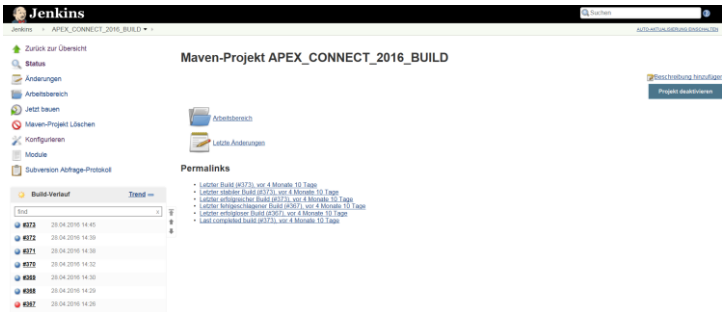

Abb. 5: Maven-Projekt in Jenkins

Maven als Build-Werkzeug
Maven lässt sich sehr einfach in Jenkins integrieren. Dafür muss lediglich über die Jenkins Plugin Verwaltung das Plugin „Maven Integration Plugin“ installiert werden. Anschließend lässt sich in Jenkins leicht ein neues Maven Projekt anlegen. Der eigentliche Maven Build Prozess wird typischerweise in einer pom.xml Konfigurationsdatei beschrieben. Für den Build Prozess von SQL/PLSQL Code bietet sich das Maven Plugin „sql-maven-plugin“ an. Durch dieses Plugin lassen sich sehr einfach in Quelldateien gespeicherte SQL bzw. PLSQL Befehle ausführen. Eine beispielhafte Konfiguration dieses Plugins ist in dem folgenden Listing dargestellt:

```
<plugin>

 <groupId>org.codehaus.mojo</groupId>
```

[1] https://jenkins.io/

```
<artifactId>sql-maven-plugin</artifactId>
<version>1.4</version>
<dependencies>
 <dependency>
  <groupId>com.oracle.jdbc</groupId>
  <artifactId>ojdbc7</artifactId>
  <version>12.1.0.1</version>
 </dependency>
</dependencies>
<configuration>
 <driver>oracle.jdbc.driver.OracleDriver</driver>
 <url>jdbc:oracle:thin:@192.168.56.102:1521:workshop</url>
 <username>ci_test</username>
 <password>ci_test</password>
 <skip>false</skip>
</configuration>
<executions>
 <execution>
  <id>test</id>
  <phase>process-resources</phase>
  <goals>
   <goal>execute</goal>
  </goals>
  <configuration>
   <autocommit>true</autocommit>
   <delimiter>/</delimiter>
   <delimiterType>row</delimiterType>
   <onError>abort</onError>
   <srcFiles>
    <srcFile>ci_test/trunk/DDL/t_benutzer.sql</srcFile>
    <srcFile>ci_test/trunk/DDL/t_benutzer_pk.sql</srcFile>
    <srcFile>ci_test/trunk/DDL/t_benutzer_uk1.sql</srcFile>
    <srcFile>ci_test/trunk/DDL/seq_t_benutzer.sql</srcFile>
```

```
      <srcFile>ci_test/trunk/PLSQL/benutzerverwaltung.pks.sql</srcFile>
      <srcFile>ci_test/trunk/PLSQL/benutzerverwaltung.pkb.sql</srcFile>
      </srcFiles>
     </configuration>
    </execution>
   </executions>
</plugin>
```

Da die Verbindung zur Datenbank über eine jdbc Verbindung erfolgt, muss ein entsprechender Treiber vorhanden sein. Die Verbindungsinformationen werden über einen einfachen Verbindungsstring, im XML-Element „URL“, angegeben. Der Konfigurationsteil „configuration“ im „executions“ Bereich listet die Quelldateien auf, die innerhalb des Build Prozesses ausgeführt werden sollen. Ist der Build Prozess erfolgreich, ist dies innerhalb des Jenkins Build Verlaufs (siehe Abbildung 5) durch einen blauen Punkt ersichtlich. Ist der Build Prozess fehlerhaft, wird dies durch einen roten Punkt kenntlich gemacht. Zudem gibt Jenkins eine detaillierte Auskunft über den aufgetretenen Fehler. Diese Integration wird durch das zuvor erwähnte „sql-maven-plugin“ Plugin ermöglicht. Auch wenn diese Art der Codeausführung sehr komfortabel ist, ergeben sich die folgenden Probleme:

Wenn existierende Datenbankobjekte wie z.B. Tabellen bei jedem Build erneut erzeugt werden, kommt es zu einem Fehler, wenn diese Objekte bereits existieren.
Fehlerhafte Prozeduren/Funktionen/Packages werden von der Datenbank trotz Fehler kompiliert (auch wenn als fehlerhaft) und Jenkins erkennt das Build aufgrund dessen als erfolgreich. Abbildung 6 zeigt ein eigentlich fehlerhaftes Package (benutzerverwaltung), welches von Jenkins aber als fehlerfrei dargestellt wird.

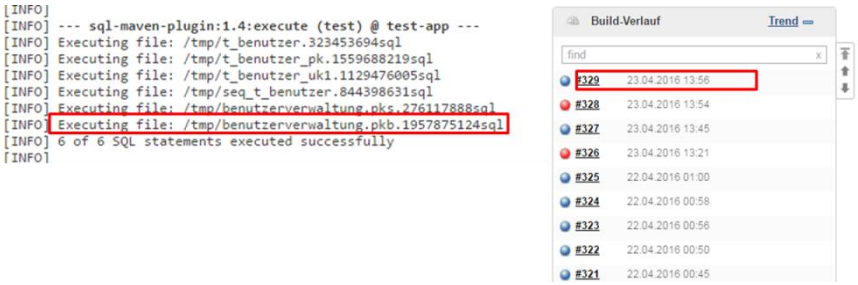

Abb. 6: Fehlerhaften Packages werden durch den Build mit „sql-maven-plugin“ in Jenkins als korrekt erkannt

Das erste Problem würde sich relativ einfach lösen lassen, wenn vor jedem Build alle Datenbankobjekte gelöscht werden. Auch wenn dies in CI Umgebungen vertretbar ist, könnten die gleichen Build Skripte nicht auf anderen Instanzen (Test,QA,etc.) verwendet werden. Daher sollten alle DML Operationen durch ein „EXECUTE IMMEDIATE“, wie in dem folgenden Listing dargestellt, erstellt werden:

```
BEGIN


 EXECUTE IMMEDIATE

  'CREATE TABLE "T_BENUTZER"

   ("BENUTZER_ID" NUMBER,
```

```
    "VORNAME" VARCHAR2(200 BYTE),
    "NACHNAME" VARCHAR2(600 BYTE)
   )';
EXCEPTION
 WHEN OTHERS THEN
  IF SQLCODE = -955 THEN
   NULL;
  ELSE
   RAISE;
  END IF;
END;
```

Durch dieses Vorgehen wird mittels des Exception-Blocks der Fehler abgefangen, der geworfen wird, falls ein Objekt bereits existiert (ORA-00955). Das zweite Problem lässt sich mit dem „sql-maven-plugin“ nicht lösen. Um auch fehlerhafte Prozeduren/Funktionen/Packages in Jenkins als fehlerhaft darzustellen, kann das Maven Plugin „exec-maven-plugin“ in Kombination mit SQL*PLUS verwendet werden. Mit diesem Plugin lassen sich Programme auf Betriebssystemebene aufrufen. Im Folgenden eine beispielhafte Maven Konfiguration:

```
<plugin>
 <groupId>org.codehaus.mojo</groupId>
 <artifactId>exec-maven-plugin</artifactId>
 <version>1.4.0</version>
 <executions>
  <execution>
   <phase>process-resources</phase>
   <goals>
    <goal>exec</goal>
   </goals>
  </execution>
 </executions>
 <configuration>
  <executable>sqlplus64</executable>
  <environmentVariables>
   <LD_LIBRARY_PATH>/usr/lib/oracle/12.1/client64/lib/</LD_LIBRARY_PATH>
  </environmentVariables>
```

```
  <arguments>
   <argument>ci_test/ci_test@192.168.56.102:1521/workshop</argument>
<argument>@install_database_objects/install_database_objects.sql</argument>
  </arguments>
 </configuration>
</plugin>
```

Das im nächsten Listing dargestellte sql Skript „install_database_objects.sql“ dient dabei als aufrufendes Skript für alle weiteren SQL Skripte und wird mittel SQL*PLUS ausgeführt.

```
WHENEVER SQLERROR EXIT SQL.SQLCODE
set define off
@install_database_objects/install_packages.sql
@install_database_objects/install_apex.sql
show errors
declare
 l_count_errors number;
begin
 select count(*)
  into  l_count_errors
  from  all_errors;
 if l_count_errors > 0 then
  raise_application_error(-20001,'Fehler beim komilieren einer PL/SQL
Prozedur');
 end if;
end;
exit 0;
```

In diesem Skript wird durch die dargestellte anonyme Funktion im Fehlerfall ein Anwendungsfehler (raise_application_error) geworfen. Der eigentliche Trick besteht aber in dem SQL*PLUS Kommando „WHENEVER SQLERROR EXIT SQL.SQLCODE“. Durch dieses Kommando wird SQL*PLUS mit einem Fehler beendet, was von Jenkins auch als Fehler erkannt wird. Abbildung 7 zeigt das nun als fehlerhaft erkannte Paket „benutzerverwaltung“.

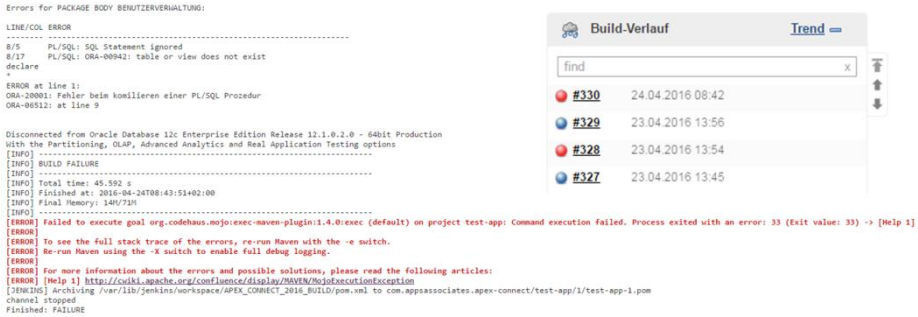

*Abb. 7: Durch den SQL*PLUS basierten Build werden fehlerhafte Packages durch Jenkins als fehlerhaft erkannt*

Neben den Datenbankobjekten lassen sich auch ganze APEX Anwendungen, einzelne APEX Seiten etc. als SQL Datei exportieren und in das Versionskontrollsystem übertragen. Die Installation der APEX Objekte kann wie für Datenbankobjekte mittels SQL*PLUS in der CI Umgebung erfolgen. Das folgende Listing zeigt ein Beispiel dafür:

```
begin

 apex_application_install.set_workspace_id(5512166499766120);

 apex_application_install.set_application_id(102);

 apex_application_install.set_schema('CI_TEST');

 apex_application_install.generate_offset;

end;

@ci_test/trunk/APEX/CI_TEST.sql

@ci_test/trunk/APEX/PAGE_1.sql

@ci_test/trunk/APEX/PAGE_2.sql

@ci_test/trunk/APEX/PAGE_3.sql

@ci_test/trunk/APEX/PAGE_101.sql
```

In der anonymen Funktion wird zunächst die Workspace ID, die Applikations ID und das Schema der zu installierenden Anwendung durch das Package „apex_application_install“ gesetzt. Des Weiteren wird mit „generate_offset“ ein Offset für die internen IDs aller APEX Objekte erzeugt. Anschließend lassen sich die einzelnen APEX Objekte durch einen einfachen Aufruf der entsprechenden SQL Skripte einfach installieren.

RSpec als Testwerkzeug

Als Testwerkzeug eignet sich für APEX Projekte RSpec[2], ein Behavior Driven Development (BDD) Framework für Ruby[3]. Mittels RSpec lassen sich erwartete Verhaltensweisen der zu testenden Software auf Klassen, Methoden- oder Modulebene in „describe“ Blöcken beschreiben. Die „describe“ Blöcke können weiter in „context“ Blöcke strukturiert werden. Ein „context“ Block beschreibt, wie der Name schon sagt, einen bestimmten Kontext, in welchem der Test ausgeführt wird. So kann es beispielsweise einen „describe“ Block für eine Methode geben. In diesem Block lassen sich nun durch „context“ Blöcke, z.B. durch Variablenbelegungen, verschiedenartige Ausführungsbedingungen erzeugen. Die eigentlichen Tests werden in „it“ Blöcken beschrieben. Jeder „it“ Block stellt damit einen konkreten Testfall dar.

[2] http://rspec.info/

[3] https://www.ruby-lang.org/de/

Mit dem „ruby-plsql“ Plugin steht des Weiteren eine API innerhalb von Ruby für SQL/PLSQL zur Verfügung. Mit dieser API lassen sich sehr einfach aus Ruby heraus SQL bzw. PLSQL Befehle, wie z.B. Funktionsaufrufe, ausführen. Insbesondere kann „ruby-plsql“ innerhalb von RSpec verwendet werden. Die Installation von Rspec und „ruby-plsql“ kann einfach mit RubyGems, dem Ruby Paketsystem, erfolgen. Beides lässt sich mit dem Paket „ruby-plsql-spec“ installieren. Eine Ruby Installation wird an dieser Stelle vorausgesetzt. Das folgende Listing zeigt ein Beispiel für den Test einer PLSQL Funktion:

```
require_relative "../spec_helper.rb"

describe "erstelle_benutzer" do

 it "soll Benutzer einfuegen" do

  expect(plsql.benutzerverwaltung.erstelle_benutzer("Sven","Boettcher")).to
equal(1)

  plsql.t_benutzer.delete(:vorname => 'Sven' , :nachname => 'Boettcher')

  plsql.commit

 end



 it "soll Benutzer nicht einfuegen" do

  benutzer = {:benutzer_id => plsql.seq_t_benutzer.nextval, :vorname =>
'Sven', :nachname => 'Boettcher'}

  plsql.t_benutzer.insert benutzer

  expect(plsql.benutzerverwaltung.erstelle_benutzer("Sven","Boettcher")).to
equal(0)

  plsql.t_benutzer.delete(:vorname => 'Sven' , :nachname => 'Boettcher')

  plsql.commit

 end

end
```

Im dem Beispiel wird eine Funktion zum Einfügen eines Benutzers getestet. Die Funktion liefert eine 1, falls der Benutzer erfolgreich in der Datenbank gespeichert wurde, und eine 0 im Fehlerfall zurück. Ein Benutzer soll eindeutig durch seinen Vor- und Nachnamen beschrieben werden. Das erfolgreiche Einfügen eines Benutzers wird in dem „it“ Block „soll Benutzer einfuegen“ getestet. Das erwartete Ergebnis wird durch den „expect“ Aufruf beschrieben. Dieser nimmt als Parameter die Rückgabe der über „ruby-plsql“ aufgerufenen PLSQL Funktion „benutzerverwaltung.erstelle_benutzer“ entgegen. Nur wenn der von der Funktion zurückgegebene Wert gleich 1 ist, ist der Test erfolgreich. Nachdem der Test ausgeführt wurde, wird der angelegte Benutzer aus der Datenbank entfernt. Der Test „soll Benutzer nicht einfuegen“ testet den Fehlerfall. Dafür wird zunächst ein Benutzer in der Datenbank angelegt und anschließend wird innerhalb des „expect“ Aufrufs versucht, diesen erneut anzulegen. In diesem Fall ist der Test nur dann erfolgreich, wenn die aufgerufene PLSQL Funktion eine 0 zurückliefert (d.h. der Fehler wurde abgefangen) und damit die Rückgabe dem erwarteten Ergebnis entspricht.

Die Einbindung von „ruby-plsql-spec“ in Jenkins erfolgt über den Bau eines „Free-Style“ Softwareprojektes mit dem Build Verfahren „Shell ausführen“. In diesem Build Verfahren lassen sich

beliebige Shell Programme, wie in diesem Beispiel RSpec, ausführen. Der Aufruf von RSpec kann folgendermaßen realisiert werden:

```
#!/bin/bash

source $HOME/.rvm/scripts/rvm

ruby --version

cd ci_test/tests/unit_tests

export PLSQL_COVERAGE=coverage

rspec -f RspecJunitFormatter -o test.xml --failure-exit-code 0
```

Durch den Aufruf von rspec werden alle Testfälle in dem Ordner ci_test/tests/unit_tests/spec ausgeführt. Die Option „–f RspecJunitFormatter“ formatiert die Ausgabe bzw. die Testergebnisse im JUnit Format, welche in der Datei test.xml gespeichert werden. Aufgrund dieser Formatierungsmöglichkeit lassen sich die Testergebnisse von RSpec einfach in Jenkins einbinden. Abbildung 8 zeigt ein Beispiel, in dem die beiden oben beschriebenen Tests erfolgreich waren.

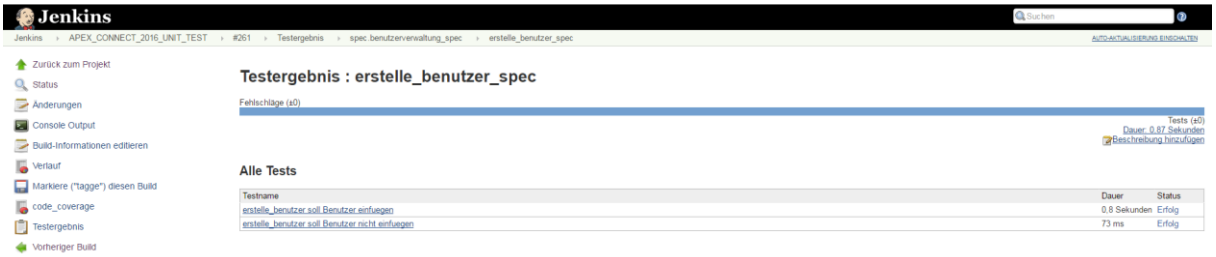

Abb. 8: Ergebnisse der RSpec basierten Funktions-Tests in Jenkins

Für den Test der APEX Anwendung lässt sich Selenium einsetzen. Selenium ist ein Tool für die Browserautomatisierung und kann zum Testen von Webanwendungen verwendet werden. Während Selenium eigentlich ein Firefox Plugin ist, steht mit dem Selenium WebDriver eine API zur Verfügung, für die es ein Ruby Plugin „selenium-webdriver“ gibt. Das Plugin lässt sich einfach mittels RubyGems installieren und ermöglicht die Browserautomatisierung mittels Ruby und RSpec. Das bedeutet, dass APEX Anwendungen ebenfalls mit RSpec getestet werden können. Durch das „ruby-plsql“ Plugin lassen sich zum einen automatisiert bestimmte Kontexte in der Datenbank erzeugen. Zum anderen lässt sich automatisch testen, ob in APEX Masken eingegeben Daten erwartungsgemäß in der Datenbank gespeichert werden. Die Einbindung der RSpec Testskripte in Jenkins erfolgt analog zu dem oben beschriebenen Test einer PLSQL Funktion. Ein Beispiel für einen Test einer einfachen APEX Maske zum Speichern eines Benutzers in die Datenbank ist im Folgenden dargestellt:

```
require "selenium-webdriver"

require_relative "spec_helper.rb"

describe "Benutzerverwaltung" do

 before(:all) do

  @driver = Selenium::WebDriver.for(:firefox)

 end
```

```
 before(:each) do
  @url = "http://192.168.56.102:8080/ords/f?p=102"
  @driver.navigate.to(@url)
  @driver.find_element(:id, "P101_USERNAME").clear
  @driver.find_element(:id, "P101_USERNAME").send_keys("admin")
  @driver.find_element(:id, "P101_PASSWORD").send_keys("Oracle1234!")
  sleep 2
  @driver.find_element(:xpath,"//button[@type='button']").click
  @url = @driver.current_url
 End

 after(:all) do
  @driver.quit
 end

 it "soll Benutzer einfuegen" do
  @url = @url.gsub("f?p=102:1","f?p=102:2")
  @driver.navigate.to(@url)
  @driver.find_element(:id,"P2_VORNAME").send_keys("Sven")
  @driver.find_element(:id,"P2_NACHNAME").send_keys("Boettcher")
  sleep 2
  @driver.find_element(:id,"speichern").click
  expect(plsql.t_benutzer.count("WHERE vorname = :vorname and nachname =
:nachname", "Sven",
   "Boettcher")).to eq(1)
  plsql.t_benutzer.delete(:vorname => 'Sven' , :nachname => 'Boettcher')
  plsql.commit
 end
end
```

Der dargestellte Test entspricht im Wesentlichen dem oben beschriebenen Test zum Einfügen eines Benutzers in die Datenbank mittels einer PLSQL Funktion, nur dass die Funktion nicht direkt, sondern über eine APEX Maske aufgerufen wird. In Zeile 5 wird zunächst ein Treiber für Firefox erzeugt. D.h. der Test erfolgt mittels Firefox. Des Weiteren existieren Treiber für Chrome und den Internet Explorer. Anschließend wird in Zeile 9 zu der APEX Anwendung navigiert und in den Zeilen 10 bis 14 erfolgt die Anmeldung. Der eigentliche Test beginnt in Zeile 20. Nachdem in Zeile 22 auf die Seite 2 der APEX Anwendung navigiert wurde, werden in den Zeilen 22 und 23 die APEX Items P2_VORNAME und P2_NACHNAME mit Werten belegt und in Zeile 26 auf den „Speichern“ Button

geklickt. Der eigentliche Test erfolgt in Zeile 26. Hier wird mittels ruby-plsql zunächst die Anzahl der Vorkommen des zuvor eingefügten Benutzers ermittelt. Ein Vergleich des Ergebnisses mit dem erwarteten Wert erfolgt, wie bereits oben beschrieben, durch „expect“. Das Ergebnis des Tests lässt sich entsprechend dem Funktionstest in Jenkins veröffentlichen.

Kontaktadresse:
Sven Böttcher
Apps Associates GmbH
Flughafenring 11
D-44319 Dortmund

Telefon:	+49 (0) 231 2222 79 34
Fax:	+49 (0) 231 2222 79 79
E-Mail	sven.boettcher@appsassociates.com
Internet:	www.appsassociates.de