Aber das hat gestern noch funktioniert! Testing mit utPLSQL

Samuel Nitsche, Smart Enterprise Solutions

Automatisierte Selbst-Tests sind inzwischen aus der Applikations- und Webentwicklung nicht mehr wegzudenken, dennoch werden sie in der Datenbankwelt nur zögerlich eingesetzt. Dieser Artikel beschreibt, wie einfach Sie mit utPLSQL Version 3 *(siehe unter https://utplsql.org)* ins Entwickeln von Selbst-Tests einsteigen können und welche Vorteile dies für Ihr Projekt und Ihre Software hat.

„Aber das hat gestern noch funktioniert!"

Dieser Ausspruch dürfte den meisten Entwickler*innen und wahrscheinlich auch einigen Benutzer*innen bekannt vorkommen. Was ist nur über Nacht passiert, dass mein Interface plötzlich nicht mehr funktioniert und dies natürlich direkt vom Kunden gemeldet wird?

Nach kurzem Nachdenken erinnern wir uns daran, dass wir die View `v_star-wars_characters` gestern kurz vor Feierabend – auf Kundenwunsch hin – noch um eine neue Spalte erweitert haben. Nichts Besonderes, aber ganz dringend: lediglich ein weiteres Feld, das Informationen darüber enthält, in welchen Filmen die jeweilige Person vorkommt.

ID	NAME	EPISODES (neu)
1	Darth Vader	3,4,5,6
2	Luke Skywalker	4,5,6,7,8,9
3	Rey	7,8,9

Ein einfaches: `create or replace view`. Anschließend haben wir natürlich per `select` getestet, ob das neue Feld auch korrekt geliefert wird. Doch wenn nun versucht wird, über die Applikation den Namen einer Person zu ändern, meldet die Datenbank den folgenden Fehler *(siehe Listing 1)*.

Vielleicht haben Sie es schon erraten, vielleicht sind Sie selbst schon über diese Eigenschaft der Oracle-Datenbank gestolpert: Beim Ersetzen einer View geht der zugehörige Instead-Of-Trigger verloren. Eine kleine, simple Änderung mit großen und ärgerlichen Auswirkungen.

Wie hätte man diesen Fehler verhindern können?

„Besser testen" ist hier natürlich die naheliegende Antwort und relativ schnell werden wir bei Checklisten und standardisierten Testprotokollen landen, mit denen sicher-

gestellt werden kann, dass eben wirklich alles Entscheidende geprüft wird. Nach jeder Änderung. An dieser Stelle werden automatisierte Selbst-Tests interessant, denn sie sind genau das: eine Reihe von exakt definierten, standardisierten „Checks" einer bestimmten Funktionalität – zum Beispiel, ob ein Update auf eine View möglich ist.

Einen solchen Test können wir mit ganz normalen PL/SQL-Bordmitteln umsetzen *(siehe Listing 2)*. Diesen Block könnten wir jetzt als Skript speichern und nach jedem Update ausführen. Allerdings wird dies mit der Zeit relativ unübersichtlich, weshalb wir stattdessen das freie Open Source Framework utPLSQL benutzen, das uns einiges an Arbeit abnimmt.

utPLSQL installieren

Um utPLSQL nutzen zu können, muss das Framework zunächst in der Datenbank installiert werden.

Dazu laden Sie am besten die aktuelle Version von GitHub herunter *(siehe unter https://github.com/utPLSQL/utPLSQL/releases/latest)*. Im Ordner „`source`" finden sich vorbereitete Skripte, mit denen sich utPLSQL problemlos installieren lässt:

- „`install_headless.sql`", um eine Standardinstallation ins Schema „ut3" mit Public Synonymen durchzuführen (erfordert SYS-user)
- „install.sql", um utPLSQL in ein anderes Schema zu installieren. In diesem Fall muss zusätzlich die Nutzung der utPLSQL-Methoden erlaubt werden:
  - Mittels „`create_synonyms_and_grants_for_public.sql`„ für alle Benutzer*innen
  - Mittels "`create_user_grants.sql`" und "`create_user_synonyms.sql`" für eine bestimmte Benutzer*in

Eine detaillierte Installationsanleitung finden Sie unter *http://utplsql.org/utPLSQL/latest/userguide/install.html* oder im Verzeichnis *docs/userguide/install.html* der heruntergeladenen ZIP-Archivs.

Ein erster Test mit utPLSQL

Während die meisten PL/SQL-Entwickler eher die prozedurale Programmierung gewohnt sind und stark auf Packages setzen, benutzt utPLSQL die objektorientierten Funktionen der Oracle-Datenbank und bietet damit eine „fluent" API an. Das mag im ersten Moment fremd erscheinen, doch Sie werden sich schnell daran gewöhnen und diesen Ansatz eventuell schätzen lernen.

utPLSQL bietet uns nach der Installation einige sehr hilfreiche Public-Methoden an, allen voran die sogenannten „Expectations" *(siehe Abbildung 1)*.

Das zweite Kernstück des Frameworks sind sogenannte „Annotations", mithilfe derer wir ganz normale PL/SQL Packages in Test-Suites umwandeln können *(siehe Listing 3)*.

`%suite` ist dabei die einzige Annotation, die absolut erforderlich ist. Sie signalisiert utPLSQL, dass es sich bei dem Package um eine Test-Suite handelt. Mit %test markieren wir die nachfolgende Prozedur als Test, den wir nun wie gewohnt im Package-Body implementieren können *(siehe Listing 4)*.

Genau wie im reinen PL/SQL-Beispiel führen wir ein Update auf die View aus. Anschließend selektieren wir das aktualisierte Feld und vergleichen das Ergebnis mit dem erwarteten Wert mittels utPLSQL-Expectation.

Da wir in einem Testszenario meist nicht genau wissen, ob und welche Daten in einer Tabelle existieren, stellen wir zunächst sicher, dass wir auf jeden Fall Daten haben, die wir aktualisieren können. Hier machen wir uns die Tatsache zunutze, dass der Primärschlüssel der meisten Tabellen ein INTEGER-Wert ist, die zugehörige Sequence oder Identity aber in den meisten Fällen bei 1 beginnt. Wir können in diesem Fall negative IDs für unsere Testdaten benutzen, ohne dass diese mit eventuell existierenden Einträgen kollidieren.

Nun können wir den Test mit der ut.run-Methode durchführen *(siehe Listing 5)*.

Wie erwartet, schlägt auch dieser Test fehl und utPLSQL gibt uns gleich den kompletten Stacktrace.

```
„ORA-01733: Virtuelle Spalte hier nicht zulässig"
```

Listing 1 - Fehlermeldung

```
declare
   l_name varchar2(2000);
begin
   update v_starwars_characters set name = 'Anakin Skywalker' where id
= 1;
   select name into l_name
       from v_starwars_characters where id = 1;

   if ( l_name <> 'Anakin Skywalker') then
       raise_application_error(-20000, 'Update did not work!');
   end if;
end;
```

Listing 2: Einfacher Test mit PL/SQL-Bordmitteln

```
ut.expect(actualValue)
    .to_equal(expectedValue)
    .to_be_greater_than(value)
    .to_be_between(min, max)
    .to_be_like('%partialString%')
    .not_to_be_null()
    .not_to_be_less_than(value)
    …
```

Abbildung 1: utPLSQL Expectations (Quelle © Samuel Nitsche). Für PL/SQL ist die objektorientierte, „fluent"-Syntax etwas ungewohnt.

```
create or replace package ut_v_starwars_characters as
  -- %suite(View: V_STARWARS_CHARACTERS)

  -- %test(Update character-name via view)
  procedure update_name;
end;
```

Listing 3: utPLSQL Test-Suite Header mit Annotations %suite und %test – diese werden vom Framework geparst und interpretiert, allerdings nur in Package-Headern

```
create or replace package body ut_v_starwars_characters as
  procedure update_name
  as
      l_actual_name v_starwars_characters.name%type;
  begin
      -- Arrange: Setup test-data
      insert into star_wars_characters (id, name) values (-1, 'Test-Char');

      -- Act: Do the actual update
      update v_starwars_characters set name = 'Darth utPLSQL' where id = -1;

      -- Assert: Check the output
      select name into l_actual_name from v_starwars_characters where id = -1;
      ut.expect(l_actual_name).to_equal('Darth utPLSQL');
  end;
end;
```

Listing 4: Implementierung des ersten utPLSQL-Tests

```
set serveroutput on
call ut.run('ut_v_starwars_characters');
View: V_STARWARS_CHARACTERS
  Update character-name via view [,002 sec] (FAILED - 1)

Failures:

  1) update_name
      ORA-01732: Datenmanipulationsoperation auf dieser View nicht zulässig
      ORA-06512: in "SITHDB.UT_V_STARWARS_CHARACTERS", Zeile 10
      ORA-06512: in "SITHDB.UT_V_STARWARS_CHARACTERS", Zeile 10
      ORA-06512: in Zeile 6
Finished in ,002346 seconds
1 tests, 0 failed, 1 errored, 0 disabled, 0 warning(s)
```

Listing 5: Ergebnis des Unit-Tests

```
create or replace trigger save_v_starwars_characters
  instead of update on v_starwars_characters
  for each row
  begin
      null;
  end;
```

Listing 6: Instead-Of-Trigger ohne Funktionalität

Interessant ist auch, dass wir in der abschließenden Zusammenfassung keinen „failed"-, sondern einen „errored"-Test angezeigt bekommen. Dieser Test hat eine ORA-Exception verursacht, die das Framework für uns abgefangen und dokumentiert hat. Hätten wir noch weitere Tests, würden diese trotzdem durchgeführt werden.

Nun können wir uns um das eigentliche Problem kümmern und den verloren gegangenen Instead-Of-Trigger wieder einspielen (zu Demonstrationszwecken zunächst ohne jede Funktionalität) *(siehe Listing 6)*.

Führen wir nun unsere Test-Suite abermals durch, erhalten wir ein etwas anderes Ergebnis *(siehe Listing 7)*.

Der Test schlägt noch immer fehl, aber dieses Mal ist es tatsächlich unsere Expectation, die den Fehler verursacht, und kein ORA-Fehler. Die Fehlerausgabe sagt uns auch sehr genau, was falsch lief und in welcher Zeile unseres Tests das Problem aufgetreten ist.

Nun implementieren wir den Trigger vollständig und führen anschließend wiederum unsere Test-Suite durch:

Unsere View funktioniert wieder wie erwartet – und das nächste Mal, wenn der Trigger verloren geht, werden wir es merken.

Vorteile automatisierter Tests

Da Sie diesen Artikel lesen, ist die Wahrscheinlichkeit relativ hoch, dass Sie nicht erst vom Nutzen automatisierter Selbst-Tests überzeugt werden müssen. Dennoch möchte ich kurz einige Vorteile auflisten, die automatisierte Tests in verschiedener Hinsicht bieten.

- Automatisierte Tests sind „change detectors", die uns bei Änderung bestehender Funktionalität warnen (unabhängig davon, ob diese Änderung gewollt ist oder – wie in unserem Beispiel – ungewollt)
- Sie sind transportabel und beliebig oft (nahezu kostenlos) auf unterschiedlichen Systemen wiederholbar
- Falls sie implementiert wurden, um Bugfixes zu bestätigen, schließen sie einmal aufgetretene Fehler aus
- Sie können als Bestätigung dienen, dass vereinbarte Anforderungen an die Software erfüllt sind

Diese Vorteile gelten für alle automatisierten Tests gleichermaßen. Wenn wir wie in diesem Fall von Tests ausgehen, die von den Entwickler*innen selbst im Zuge des Entwicklungsprozesses geschrieben werden, kommen noch einige weitere Vorteile dazu:

- Die Erstellung von Tests kann dabei helfen, den Fokus vom **Wie** auf das **Was** zu verlagern und eine Funktionalität aus unterschiedlichen Perspektiven zu betrachten
- Gut und verständlich geschriebene Tests können als Code-Beispiel und

Anders als bei sogenannten „fail-fast"-Frameworks wertet utPLSQL alle Expectations eines Tests aus und bricht nicht sofort nach dem ersten Fehlschlag den Test ab. Alle Fehlschläge werden gesammelt in das Testergebnis aufgenommen und untereinander dargestellt.

Dokumentation dafür dienen, wie eine Funktionalität zu verwenden ist
- Selbst-Tests regen dazu an, „einfachere" Programmierkonstrukte zu verwenden, da sich diese leichter testen lassen – was wiederum positive Auswirkungen auf die Wartbarkeit des Codes hat

Der aus meiner Sicht jedoch wichtigste Vorteil, den eine solide, automatisierte Testbasis bietet, ist, dass sie die Voraussetzung für die Entwickler*innen schafft, stetig und selbstbewusst den eigenen Quellcode zu verbessern, also ständiges Refactoring zu betreiben.

```
call ut.run();
View: V_STARWARS_CHARACTERS
   Update character-name via view [,409 sec] (FAILED - 1)

Failures:

   1) update_name
      Actual: 'Test-Char' (varchar2) was expected to equal: 'Darth ut-
PLSQL' (varchar2)
      at "SITHDB.UT_V_STARWARS_CHARACTERS.UPDATE_NAME", line 14 ut.ex-
pect(l_actual_name).to_equal('Darth utPLSQL');


Finished in ,41099 seconds
1 tests, 1 failed, 0 errored, 0 disabled, 0 warning(s)
```

Listing 7: Ergebnis des Unit-Tests mit „failed" statt „errored"

```
create or replace trigger save_v_starwars_characters
   instead of update on v_starwars_characters
   for each row
   begin
       update star_wars_characters
           set name = :new.name
           where id = :new.id;
   end;
/

call ut.run();
View: V_STARWARS_CHARACTERS
   Update character-name via view [,004 sec]

Finished in ,005951 seconds
1 tests, 0 failed, 0 errored, 0 disabled, 0 warning(s)
```

Listing 8: Korrektur des Triggers und erfolgreicher Test

```
create or replace package ut_v_starwars_characters as
   -- %suite(View: V_STARWARS_CHARACTERS)

   -- %beforeall
   procedure setup_test_data;

   -- %test(Update character-name via view)
   procedure update_name;

   -- %test(View returns correct list of episodes)
   procedure return_list_of_episodes;
end;
```

Listing 9: Package-Header mit neuem Test und weiteren Annotations

Ein weiterer Test und mehr Annotations

Eine Funktionalität unserer View haben wir abgesichert, doch gerade die neue Spalte enthält ein Stück Logik, bei dem es sich durchaus lohnt, das Verhalten durch einen automatisierten Test abzusichern (insbesondere dann, wenn wir davon ausgehen, dass wir die Funktionalität in Zukunft vielleicht noch mehrfach ändern oder erweitern werden).

Wir definieren also einen weiteren Test und nutzen gleich noch eine weitere utPLSQL-Annotation:

Die %beforeall-Annotation sorgt dafür, dass diese Prozedur einmalig pro Test-Suite ausgeführt wird, und zwar vor allen Tests.

Der Nutzen wird klarer, wenn wir uns die Implementierung ansehen:

`setup_test_data` stellt nun die Situation her, die wir für die Ausführung beider Tests benötigen: Ein Eintrag in der Tabelle `star_wars_characters` sowie mehrere Einträge in der Tabelle `appearance_in_episode`. Diese beiden Tabellen bilden die Grundlage der View, die wir testen möchten.

Die Tests selbst sind nun sehr einfach zu lesen und zu verstehen – zur Verbesserung der Lesbarkeit haben wir zusätzlich noch die Hilfsfunktion `get_view_row()` hinzugefügt, die einfach die komplette View-Zeile zurückgibt. Für den Check verwenden wir wiederum die `to_equal`-Expectation.

Vielleicht fragen Sie sich schon die ganze Zeit, was mit den ganzen Testdaten passiert, die wir anlegen?

utPLSQL arbeitet im Standard-Modus mit Savepoints und Rollbacks. Vor jeder Suite sowie vor jedem Test wird jeweils ein Savepoint gesetzt, zu dem nach Abschluss des Tests beziehungsweise der Suite zurückgerollt wird. Das bedeutet, dass die Testdaten, die wir anlegen, sowie alle anderen Änderungen an Daten am Ende des Tests wieder verschwunden sind.

Das bedeutet natürlich auch, dass wir in diesem Modus nur Funktionen testen können, die keine Transaktionskontrolle wie

`commit` und `rollback` oder DDL enthalten. Auch diese Szenarien können natürlich mit utPLSQL-Tests abgesichert werden. In diesem Fall fügt man unterhalb von `%suite` die Annotation `%rollback(manual)` ein, muss sich dann aber selbst darum kümmern, dass Änderungen aufgeräumt werden (beispielsweise mittels `%afterall`).

Für alle Situationen, in denen keine Transaktionskontrolle notwendig ist, bietet der Rollback-Mechanismus von utPLSQL hingegen eine große Erleichterung.

Absichern von Sonderfällen

Was wir bisher getestet und abgesichert haben sind die offensichtlichen Dinge, die wir erwarten. Was aber passiert, wenn wir einen Star-Wars-Charakter haben, der in keinem der Filme vorkommt, beispielsweise die beliebte Ahsoka Tano aus der „The Clone Wars"-Fernsehserie?

Es ist wichtig, über die offensichtlichen Use-Cases hinaus zu denken, über das hinaus zu denken, was wir von den Benutzer*innen erwarten oder möchten. Die Wahrscheinlichkeit ist sehr hoch, dass Benutzer*innen unsere Applikation auf eine Art und Weise nutzen, die wir normalerweise nicht erwarten würden.

Lassen Sie uns also einen weiteren Test für diesen Fall schreiben *(siehe Listing 11)*.

Wenn dieser Test erfolgreich durchgeführt wird, haben wir damit zwei Dinge auf einmal bewiesen:

- Auch wenn ein Charakter nicht in den Filmen vorkommt, liefert die View eine Zeile zurück (ansonsten würde eine `NO_DATA_FOUND`-Exception ausgelöst)
- Der Wert in der Spalte `EPISODES` ist in diesem Fall `NULL`.

utPLSQL bietet eine ganze Reihe von Annotations, die es ermöglichen, setup und cleanup vom eigentlichen Test zu entkoppeln:

- %beforeall
- %beforeeach
- %beforetest
- %aftertest
- %aftereach
- %afterall

```
create or replace package body ut_v_starwars_characters as

   function get_view_row
       return v_starwars_characters%rowtype
   as
       l_result v_starwars_characters%rowtype;
   begin
       select * into l_result
           from v_starwars_characters
           where id = -1;
       return l_result;
   end;

   procedure setup_test_data
   as
   begin
       insert into star_wars_characters (id, name) values (-1, 'Test-
Char');
       insert into appearance_in_episode (character_fk, episode_no)
           values ( -1, 3 );
       insert into appearance_in_episode (character_fk, episode_no)
           values ( -1, 5 );
   end;

   procedure update_name
   as
   begin
       update v_starwars_characters set name = 'Darth utPLSQL' where id
= -1;

       ut.expect(get_view_row().name)
           .to_equal('Darth utPLSQL');
   end;

   procedure return_list_of_episodes
   as
   begin
       ut.expect(get_view_row().episodes)
           .to_equal('3,5');
   end;
end;
```

Listing 10: Implementierung der erweiterten Test-Suite

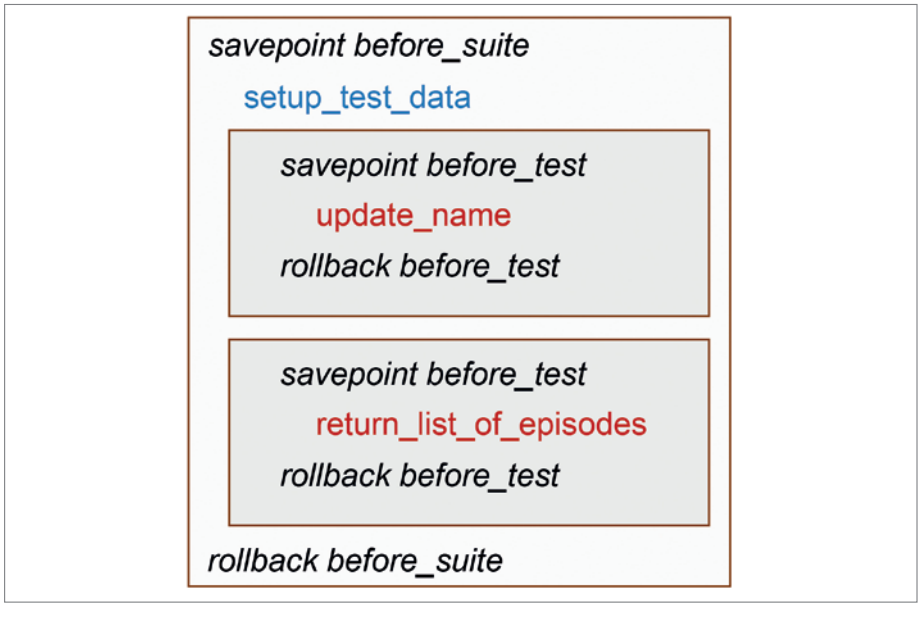

Abbildung 2: Reihenfolge und Savepoints der Test-Suite (Quelle © Samuel Nitsche)

```
-- Package-Header
…
-- %test(View returns row but empty list of episodes when character has
no appearance)
procedure return_empty_list_of_episodes;
…
-- Package-Body
…
procedure return_empty_list_of_episodes
as
begin
   delete from appearance_in_episode where character_fk = -1;

   ut.expect(get_view_row().episodes)
       .to_be_null();
end;
…
```

Listing 11: Test, um das Verhalten der View bei Nicht-Erwähnung in den Filmen abzusichern

Wir könnten nun auch noch sicherstellen, dass die View-Zeile immer noch den erwarteten Namen zurückgibt und nicht etwa `NULL`. Hier zeigt sich aber auch ein Dilemma beim Schreiben automatisierter Selbst-Tests.

Wie viele Tests sind genug?

Wie weit sollten meine Tests gehen? Sollte ich jede noch so kleine Eventualität mit einem Test absichern, in unserem Fall zum Beispiel überprüfen, dass Insert nicht erlaubt ist oder dass der Name nicht auf einen Wert geändert werden kann, der bereits existiert?

Diese Frage werden nur Sie beantworten können, denn nur Sie kennen die Umstände und Risiken Ihres Projekts und Ihrer Datenbankapplikation. Hilfreich bei der Entscheidung können die folgenden Fragen sein:

- Wie gravierend sind die Folgen, wenn sich eine bestimmte Funktionalität nicht verhält wie erwartet?
- Wie wahrscheinlich ist es, dass ein bestimmter Fall auftritt (z.B., dass ein Tabellen-Constraint versehentlich gelöscht wird und damit Mehrfach-Namen möglich werden)?
- Wie wahrscheinlich ist es, dass sich der Code dieser Funktionalität in Zukunft ändern wird und wie häufig wird dies der Fall sein?
- Wie schwierig beziehungsweise aufwendig ist es, die Funktionalität automatisiert zu testen?

Software-Entwicklung ist oftmals das Ausbalancieren von Kompromissen und genauso verhält es sich auch mit der Entwicklung automatisierter Tests. Ob und in welchem Umfang diese sinnvoll sind, hängt sehr stark von Ihren Zielen, Ihrem Entwicklungsprozess, Ihren Unternehmens- und Projektumständen ab.

Schon ein paar wenige Tests, die wichtiges bestehendes Verhalten auf einem relativ hohen Level absichern, können ungemein hilfreich sein. Wenn Sie hingegen eine Software entwickeln, die über Jahre gewartet, erweitert und verbessert werden soll, wird eine detailliertere Basis von Unit-Tests, die das ständige Refactoring des Codes ermöglichen, einen echten Mehrwert bringen und Ihre Entwicklungsgeschwindigkeit und -qualität drastisch erhöhen.

Mein Tipp: Fangen Sie klein an und probieren Sie aus. Jedes Mal, wenn ein Fehler auftritt oder gemeldet wird, müssen Sie ohnehin analysieren und versuchen, den Fehler nachzustellen. Dies können Sie in der Regel auch so tun, dass Sie die entsprechenden Voraussetzungen in Form eines Test-Setups festlegen. Wenn der Fehler dann gefunden und behoben ist, haben Sie gleich einen Test, der sicherstellt, dass dieser Fehler in Zukunft nicht wieder auftritt.

Aller Anfang ist schwer, lassen Sie sich nicht davon entmutigen. Experimentieren Sie und beobachten Sie, was Ihnen hilft. Es sind oft nicht die großen, gewaltigen Schritte, die nachhaltige Änderung bringen, sondern die kleinen, stetigen, die ein Teil des Alltags werden.

Weitere Informationen und Tools

Viele Informationen rund um utPLSQL finden Sie im „Resources"-Bereich von utplsql.org *(siehe unter https://utplsql.org/resources)*.

Mittlerweile gibt es auch eine umfangreiche Sammlung von Tools rund um utPLSQL:

- SQLDeveloper Plug-in:
  https://www.salvis.com/blog/2019/07/06/running-utplsql-tests-in-sql-developer/
  Command Line Interface:
  https://github.com/utPLSQL/utPLSQL-cli
- Maven-Plug-in:
  https://github.com/utPLSQL/utPLSQL-maven-plugin
- Demo-Projekt inklusive CI/CD-Integration in Travis:
  https://github.com/utPLSQL/utPLSQL-demo-project

Alle Codebeispiele inklusive Setup finden Sie unter:
http://bit.ly/sithdb-aber-das-hat-gestern-noch-funktioniert

Autoren-Blog:
https://cleandatabase.wordpress.com

Über den Autor

Samuel Nitsche ist ein stets neugieriger Software-Entwickler und Oracle ACE mit beinahe 20 Jahren Entwicklungserfahrung. Nebenberuflich schreibt er regelmäßig zu Datenbank-Entwicklungsthemen, präsentiert auf Meetups und Konferenzen (gerne auch in Sith-Robe) und gehört dem Kern-Entwicklungsteam hinter utPLSQL an.

Samuel Nitsche
derpesse@gmail.com