Pruebas unitarias en Postgres

Jorge Gómez Reus

25 de febrero de 2019

Índice

1. Introducción

1.1. Pruebas Unitarias

Las pruebas son una parte fundamental del desarrollo de software ya que nos permiten asegurar en parte la calidad de este. Las pruebas unitarias son un nivel de pruebas de software que involucra probar los componentes/unidades del software. **Una unidad es la parte más pequeña del software capaz de ser probada**.

Existen frameworks pruebas unitarias para el nivel de aplicaciones, en el caso de java: Junit, TestNG, entre otros, pero estos frameworks solo son del nivel de aplicación, por ende necesitamos una herramienta que nos permita hacer pruebas a un nivel más bajo en el stack de tecnologías i.e. la base de datos.

1.2. pgTAP

pgTAP es un conjunto de funciones de base de datos que nos facilitan escribir pruebas estilo TAP en scripts de psql o funciones estilo XUnit. El formato de salida estilo **TAP** nos permite obtener, analizar y reportar la información de las pruebas usando herramientas estandarizadas, como por ejemplo **pg_prove**. La gran ventaja de pgTAP es que las funciones de prueba fueron escritas en pgplsql, por no tienen el *overhead* de un driver de postgres o un ORM.
Ejemplo:

```
1 -- Comenzamos una transacción
2 BEGIN;
3 -- Seleccionamos el número de pruebas a hacer, en este caso 2
4 SELECT plan(2);
5 --Variables
```

```
\set var_id_alumno 1

-- Prueba de inserción
SELECT ok(
   insertar_alumno(:src_id, 'nombre', 'apellido_p', 'apellido_m'),
   'insertar_alumno()_debería_regresar_true'
);

-- Prueba de consulta
SELECT is(
   ARRAY(
   SELECT nb_alumno FROM tal01_alumno WHERE id_alumno = :id_alumno;
   ),
   ARRAY['character_varying'],
   'Consultamos_el_nombre_de_un_alumno'
);
--Le decimos a pgTAP que las pruebas fueron completadas,
--esto para que indique los resultados
SELECT * FROM finish();
--Hacemos un rollback de la transacción
ROLLBACK;
```

1.3. Instalación

1. git clone https://github.com/jlavallee/tap-harness-junit.git
2. git clone https://github.com/rjbs/Test-Deep.git
3. sudo cpan Module::Build
4. sudo cpan XML::Simple
5. cd Test-Deep && perl Makefile.PL && sudo make && sudo make test && sudo make install
6. cd tap-harness-junit && perl Build.PL && sudo ./Build && sudo ./Build install

1.4. ¿Qué nos permite hacer pgTAP?

- Probar existencia, tipo de lenguaje, tipo de retorno y "salud" de una función.

```
  BEGIN;
     SELECT plan(4);
     SELECT has_function(
     'public',
     'spsce_1',
     ARRAY[ 'integer', 'integer', 'integer' ],
     'Probar_la_existencia_de_la_función_spsce_1(idCiclo,_idNivel,_idGrupo)'
     );
     SELECT function_lang_is(
     'public',
     'spsce_1',
     ARRAY[ 'integer', 'integer', 'integer' ],
```

```
    'plpgsql',
    'Probar que la función spsce_1 está escrita el plpgsql'
    );
    SELECT function_returns(
    'public',
    'spsce_1',
    ARRAY[ 'integer', 'integer', 'integer' ],
    'setof tce03_grupo',
    'Probar que la función spsce_1 retorna setof tce03_grupo'
    );
    \set id_ciclo 1
    \set id_nivel 1
    \set id_grupo 1
    SELECT lives_ok(
       FORMAT(
       'SELECT * FROM public.spsce_1( %s, %s, %s)',
        :id_ciclo, :id_nivel, :id_grupo),
       'Probar que la función spsce_1 no genere error en tiempo de ejecución'
    );
    SELECT * FROM finish();
ROLLBACK;
```

- Verificar la existencia de esquemas, tablas, indices, llaves foráneas, tipos, etc.

```
BEGIN;
    SELECT plan(8);
    SELECT has_schema('public', 'Verificar que el esquema public exista');
    SELECT has_table('public', 'tal01_alumno', 'Verificar que la tabla public.
        ↪ tal01_alumno exista');
    SELECT has_sequence('public','tal01_alumno_id_alumno_seq',
       'verificar que la secuencia tal01_alumno_id_alumno_seq exista');
    SELECT has_type( 'public', 'sce_25', 'verificar que el tipo de dato sce_25
        ↪ exista' );
    SELECT has_extension('public', 'pgtap', 'verificamos que la extensión pgtap
        ↪ _exista');
    SELECT has_column('public', 'tal01_alumno', 'tx_matricula',
       'Probar que la tabla public.tal01_alumno tenga la columna tx_matricula')
          ↪ ;
    SELECT has_pk('public', 'tal01_alumno', 'Probar que la tabla public.
        ↪ tal01_alumno tenga pk');
    SELECT fk_ok(
       'public',
       'tal01_alumno',
       ARRAY['id_sexo'],
       'public',
       'tib14_sexo',
       ARRAY['id_sexo']
    );
    SELECT * FROM finish();
ROLLBACK;
```

- Probar configuraciones del gestor de base de datos.
- Probar consultas.

1.5. Muy bonito y todo. ¿Cómo se usa?

Al ser consultas se pueden ejecutar mediante un script de **psql** o directamente en una sesión. Ejecutar prueba por prueba es un dolor de cabeza, entonces por salud mental, que automáticamente se haga, ¿no?. Para la tarea de automatizarlo y darle un formato de salida adecuado existen herramientas de ejecución, tales como **pg_prove**.
Ejemplo: *pg_prove -d eld-superior-test TEST/*.sql –verbose*

- -d especifica la base de datos a usar
- Los siguientes argumentos son todos los archivos .sql que contienen los scripts de pruebas
- –verbose significa que pg_prove debe de mostrar los más que pueda de información

2. Integración con Jenkins

Con base en las metodologías de CI y CD podemos solucionarlo. Estaría genial que se pudiera ejecutar cada vez que se hiciera un commit en el **VCS**, entonces **Jenkins** u otro servidor de CI/CD sería la opción. Instrucciónes Mediante Jenkins CLI:

1. Descargar jenins CLI dando click en el link *jenkins-cli.jar* dando click en la pantalla de jenkins CLI
2. Verificar que el comando *java -jar jenkins-cli.jar -s http://$JENKINS_IP:$JENKINS_PORT/ -auth $USER:$PASSWORD list-plugins | grep junit* muestre junit en la salida, en el caso contrario:
   java -jar jenkins-cli.jar -s http://$JENKINS_IP:$JENKINS_PORT/ -auth $USER:$PASSWORD install-plugin junit
3. Modificar el archivo XML
   Archivo de Ejemplo

```
<?xml version='1.1' encoding='UTF-8'?>
<flow-definition plugin="workflow-job@2.31">
  <actions>
    <org.jenkinsci.plugins.pipeline.modeldefinition.actions.DeclarativeJobAction
        ↪ plugin="pipeline-model-definition@1.3.4.1"/>
    <org.jenkinsci.plugins.pipeline.modeldefinition.actions.
        ↪ DeclarativeJobPropertyTrackerAction plugin="pipeline-model-definition@1
        ↪ .3.4.1">
      <jobProperties/>
      <triggers/>
      <parameters/>
      <options/>
    </org.jenkinsci.plugins.pipeline.modeldefinition.actions.
        ↪ DeclarativeJobPropertyTrackerAction>
  </actions>
```

```
  <description></description>
  <keepDependencies>false</keepDependencies>
  <properties/>
  <definition class="org.jenkinsci.plugins.workflow.cps.CpsFlowDefinition" plugin=
      ↪ "workflow-cps@2.63">
    <script>pipeline {
    agent any
    stages {
       stage(&apos;Stage 1&apos;) {
          steps {
             sh &apos;pwd&apos;
             sh &apos;JUNIT_OUTPUT_FILE=reports/junit/results.xml pg_prove -d eld-
                 ↪ superior-test /home/lady/Desktop/TEST/*.sql --verbose --harness
                 ↪  TAP::Harness::JUnit&apos;
          }
       }
    }
     post {
      always {
        junit &apos;reports/junit/results.xml&apos;
      }
    }
}</script>
    <sandbox>true</sandbox>
  </definition>
  <triggers/>
  <disabled>false</disabled>
</flow-definition>
```

4. Crear un nuevo *job* con el comando:
   java -jar jenkins-cli.jar -s http://$JENKINS_IP:$JENKINS_PORT/ -auth $USER:$PASSWORD create-job $NOMBRE <ejemplo.xml

5. Crear las carpetas *reports/junit* en */var/lib/jenkins/workspace/$NOMBRE*

6. Ejecutar el nuevo **job** con el comando:
   java -jar jenkins-cli.jar -s http://$JENKINS_IP:$JENKINS_PORT/ -auth $USER:$PASSWORD build $NOMBRE