Année académique 2012/2013

TP 5 – Introduction à PL/pgSQL

1. Initialisation :

A faire : Lancer pgAdmin III

2. Le langage PL/pgSQL (Procedural Language for postgreSQL) :

La nature relationnelle, déclarative de SQL rend l'expression des requêtes très naturelle mais les applications complexes exigent plus ➔Nécessité d'étendre les fonctionnalités de SQL : PL/pgSQL est une extension procédurale.
Le langage PL/pgSQL est un langage procédural (équivalent au PL/SQL sous Oracle) et livré avec PostgreSQL.
Le langage PL/pgSQL est un langage complet avec des structures de contrôle, des boucles … avec l'avantage d'être très proche du langage SQL, et donc d'être compatible avec les types de données, les opérateurs et les fonctions SQL (ceci assure la portabilité des procédures stockées en PL/pgSQL par rapport à toutes les instances de PostgreSQL).

Pour activer PL/pgSQL pour une base de données, exécuter la requête :

```
CREATE LANGUAGE plpgsql;
```

Ou sous psql (outil en ligne de commande) avec la commande :

```
[postgres]# createlang plpgsql base
```

A faire : Activer le langage plpgsql pour la base de données RESERV.

3. Procédures stockées en PL/pgSQL :

Syntaxe d'une fonction PL/pgSQL :

```
CREATE OR REPLACE FUNCTION nom_fonction (paramétres)
  RETURNS [type | VOID]
  AS
'
     DECLARE
          Déclarations des variables ;
     BEGIN
          Instructions ;
     END;
'
LANGUAGE 'plpgsql';
```

Exemple de déclaration d'une fonction :

```
CREATE OR REPLACE FUNCTION carre (integer)
  RETURNS integer
  AS
'
  BEGIN
    RETURN $1*$1;
  END;
'
LANGUAGE 'plpgsql';
```

Appel de la fonction :

```
SELECT carre(11);
```

Fonction PL/pgSQL incluant une requête SQL :

```
CREATE OR REPLACE FUNCTION DateReser (integer)
  RETURNS DATE
  AS
'
     DECLARE
          dateres DATE;
     BEGIN
          SELECT INTO dateres date_resa
          FROM Reservation WHERE reservation_id = $1;
     /* Le résultat de la requete est stockée dans dateres*/
          RETURN dateres;
     END;
'
LANGUAGE 'plpgsql';
```

A faire : Créer puis exécuter la fonction DateReser

Déclarations de variables :

- `dateres DATE;`
- `pi CONSTANT real:=3.14;`
- `nom_etudiant varchar:="OUJDI";`

Opérations de base :

Affectation : `identifiant:= {expression | valeur};`

Structures conditionnelles :

```
IF boolean-expression THEN
    statements
```

```
END IF;

IF ... THEN ... ELSIF ... THEN ... ELSE ... END IF;
```

Structures itératives : `LOOP / WHILE / FOR`

(Se référer à l'aide de PostgreSQL)

Variable de type enregistrement (ROWTYPE) :

Il est intéressant d'utiliser les enregistrements ou tuples comme variables. Ceci se fait au niveau du bloc de déclaration.
La syntaxe est : `nom_variable nom_table%ROWTYPE;`
Ainsi on peut stocker dans cette variable un enregistrement de la table nom_table.

```
DECLARE
       monEtudiant etudiant%ROWTYPE;
       NomPrenom text;
BEGIN
     SELECT INTO monEtudiant * FROM etudiant
     WHERE etudiant_id = $1;
     NomPrenom := monetudiant.nom || '' '' ||
     monetudiant.prenom;
     RETURN NomPrenom;
```

Remarque : Remarquer que nous avons doublé ' ' en '' '' pour éviter la confusion avec le ' de code.

A faire : Tester le code précédent dans une fonction.

Alias de paramètres :

Il est parfois préférable de remplacer la référence aux paramètres ($1, $2, $3…) par des alias plus parlant. Ceci peut être réalisé dans le bloc déclaration avec la syntaxe :

```
alias1 ALIAS FOR $1;
```

A faire : En utilisant les alias écrire une fonction qui prend en argument le numéro de salle et le numéro de bâtiment puis retourne la capacité de la salle.

NOT FOUND / IS NULL :

Dans le cas de l'utilisation d'un SELECT INTO on peut écrire :

- `IF NOT FOUND THEN` … pour tester si rien n'est retourné.
- `IF nom_table.nom_colonne IS NULL THEN` … pour tester si la colonne est vide.

Appel d'une fonction par une fonction :

```
CREATE OR REPLACE FUNCTION TotalHeureReserv (integer)
  RETURNS integer
  AS
'
     DECLARE
          total integer;
     BEGIN
          SELECT sum(nombre_heures)INTO total
          FROM Reservation WHERE date_resa = DateReser($1);
          RETURN total;
     END;
'
LANGUAGE 'plpgsql';
```

Afficher un message RAISE NOTICE :

Vous pouvez afficher un message ou des valeurs :

```
CREATE OR REPLACE FUNCTION TotalHeureReservBis (integer)
  RETURNS integer
  AS
'
DECLARE
     total integer;
BEGIN
     SELECT sum(nombre_heures)INTO total
          FROM Reservation WHERE date_resa = DateReser($1);
          RAISE NOTICE ''Total Heure Réservé le : %'' ,
DateReser($1);
          RAISE NOTICE ''est : %'' , total;
          RETURN total;
     END;
'
LANGUAGE 'plpgsql';
```

Gestion des exceptions RAISE EXCEPTION :

Le mot clé RAISE permet de lancer des messages d'avertissement ou d'erreur. Dans le cas d'une erreur utiliser RAISE EXCEPTION. Ceci en plus du message arrête l'exécution de la procédure et crée une erreur.

```
CREATE OR REPLACE FUNCTION Division (numeric, numeric)
  RETURNS numeric
  AS
'
DECLARE
     quotient numeric;
BEGIN
```

```
     IF $2 = 0 THEN
          RAISE EXCEPTION ''Division par zero !!!'' ;
     END IF;
     quotient := $1 / $2;
     RETURN quotient;
     END;
'
LANGUAGE 'plpgsql';
```

A faire : En utilisant RAISE écrire une fonction qui retourne le téléphone d'un enseignant connaissant son identifiant, et qui retourne une erreur si l'enseignant n'existe pas ou que le téléphone est NULL.

4. Triggers (Déclencheurs) :

Un déclencheur ou trigger est un programme stocké dans une base de données et associé à une table de la base de données.
Un trigger est associé à un événement de type INSERT, UPDATE ou DELETE qui se produit sur cette table. Le trigger est exécuté automatiquement lorsque l'événement auquel il est attaché se produit sur la table.
Les triggers peuvent être écrits en n'importe quel langage de procédure stockée.

Syntaxe :

```
CREATE TRIGGER name { BEFORE | AFTER } { event [ OR ... ] }
    ON table [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( condition ) ]
    EXECUTE PROCEDURE function_name ( arguments )
```

Exemple : (Pris sur PostgreSQL)

```
CREATE TABLE emp (
    empname text,
    salary integer,
    last_date timestamp,
    last_user text
);

CREATE FUNCTION emp_stamp() RETURNS trigger AS
'
    BEGIN
        -- Check that empname and salary are given
        IF NEW.empname IS NULL THEN
            RAISE EXCEPTION ''empname cannot be null'';
        END IF;
        IF NEW.salary IS NULL THEN
            RAISE EXCEPTION ''% cannot have null salary'',
NEW.empname;
        END IF;

        -- Who works for us when she must pay for it?
        IF NEW.salary < 0 THEN
            RAISE EXCEPTION ''% cannot have a negative
salary'', NEW.empname;
        END IF;

        -- Remember who changed the payroll when
        NEW.last_date := current_timestamp;
        NEW.last_user := current_user;
        RETURN NEW;
    END;
'
LANGUAGE plpgsql;
```

```
CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE emp_stamp();
```

A faire : Créer une nouvelle base de données Test et exécuter les requêtes précédentes. Puis, tester chaque exception prévue dans le trigger avec INSERT INTO.

Exercice 2 :

En reprenant le même énoncé que dans l'exercice 1, écrire un trigger qui empêche la réservation d'une salle si elle est occupée.
La procédure stockée sera écrite en langage plpgsql.