PL/pgSQL

Programming Language pour *Postgres* basé sur SQL

Fabien Coelho, Claire Medrala

Mines Paris – PSL

Décembre 2023

Code applicatif ? *client vs serveur vs DB*

client	code exposé, modifiable...
serveur	langage type Python, Ruby, NodeJS
db	langage type PL/SQL

Avantages côté serveur

- même langage/environnement que l'application
- debug plus facile ?

Avantages côté DB

- code partagé par toutes les applications
- moins de traffic app/stockage
- performances accrues car requêtes parsées et préparées

Introduction à PL/pgSQL

Langage d'extension de Postgres

PL	langage de programmation côté serveur installation nécessaire : `CREATE LANGUAGE` interprété : erreurs de syntaxe possibles à l'exécution
SQL	à la base : types, expressions, requêtes inspiré du PL/SQL d'Oracle
langage	fonctions, déclarations, conditions, boucles, exceptions adapté au relationnel : retour d'un tuple, d'une relation... scalaires ou tableaux polymorphes
aussi	SQL, PL/(C Python Perl Ruby Tcl PHP R Java sh PSM...) *trusted* vs *untrusted* : protégé ou peut planter un serveur

Utilisations des langages côté DB

fonction	nouvelle accessibles en SQL `CBRT`...
aggrégation	nouvelle `SUM AVG`...
opérateur	supplémentaire `=+ **`
type/domaine	nouveaux
cast	traduction de type explicite, affectation ou implicite
conversion	de chaînes de caractères *latin1 utf8*...
trigger	actions automatiques lors d'évènements (DML ou DDL)
application	scripts spécifiques, réduction des A/R

Structure de PL/pgSQL

Entêtes

- déclaration ou remplacement d'une fonction
  important lors du développement, attention aux erreurs !
  `CREATE OR REPLACE FUNCTION`
- déclaration de la signature : nom, retour, arguments
  `log_nep(a REAL) RETURNS REAL`
- caractéristiques particulières
  `RETURNS NULL ON NULL INPUT`
- début du code
  `AS $$`

Structure de PL/pgSQL (2)

Corps

- déclaration des variables et code de la fonctions

```
BEGIN
  IF a>0 THEN
    RETURN LN(a);
  END IF;
  RAISE EXCEPTION 'argument negatif invalide';
END;
```

- fin du code, précise le langage
  `$$ LANGUAGE plpgsql;`

Utilisation de la fonction

```
SELECT log_nep(10.0); -- 2.30259
SELECT log_nep(-3.0); -- ERROR: argument negatif invalide
SELECT log_nep(NULL); -- NULL
```


Exemple PL/pgSQL


```
CREATE OR REPLACE FUNCTION
plus_un(i INTEGER) RETURNS INTEGER
RETURNS NULL ON NULL INPUT
AS $$
BEGIN
  RETURN i+1;
END;
$$ LANGUAGE plpgsql;

SELECT plus_un(3) AS "total";
```

total
4

Signature `CREATE FUNCTION/PROCEDURE`

Typage des paramètres

- type PostgreSQL des arguments et du résultat pour les fonctions
  accès aux arguments possible par leur numéro
  `is_one(INTEGER) RETURNS BOOLEAN ...`
  `add(REAL, REAL) RETURNS REAL ...`
- nommage des arguments, plus agréable
  `email(nom TEXT, domaine TEXT) RETURNS TEXT`
- types génériques scalaire ou tableau
  `last(a ANYARRAY) RETURNS ANYELEMENT`
- référence au type d'un attribut d'une relation
  `x nom_table.nom_attribut%TYPE`

Déclaration des caractéristiques des fonctions/procédures

Caractéristiques : stabilité, NULL

- unification des appels de fonctions, volatile par défaut
  - `IMMUTABLE` résultat constants selon les arguments
  - `STABLE` résultat constant dans un scan : *requête dans fonction...*
  - `VOLATILE` résultat variable `RANDOM NEXTVAL`
- gestion si arguments `NULL`
  - `STRICT` (ou `RETURNS NULL ON NULL INPUT`) ne pas appeler
  - `CALLED ON NULL INPUT` appeler, arguments gérés, par défaut

Déclaration des caractéristiques (suite)

Caractéristiques : permissions, divers, appel

- droits lors d'un appel
  `SECURITY DEFINER/INVOKER` selon
- autres détails :
  - `PARALLEL` selon `SAFE, RESTRICTED, UNSAFE`
  - `LEAKPROOF` effets de bord...
  - `WINDOW` fonction de fenêtre (sorte d'aggrégation...)
  - `COST` coût d'appel, utilisé par l'optimiseur
  - `ROWS` nombre de tuples retournés
- appel dans une expression (fonction) ou `CALL` (procédure)

Caractéristiques d'une fonction

- informations importantes pour l'optimiseur de requêtes
- valeurs par défaut conservatives
  `VOLATILE` `CALLED ON NULL INPUT` `SECURITY INVOKER`

```
CREATE FUNCTION nrows(name TEXT) RETURNS INTEGER
  STRICT STABLE ...

CREATE FUNCTION chpass(old TEXT, new TEXT) RETURNS BOOLEAN
  STRICT STABLE SECURITY DEFINER ...

CREATE FUNCTION concatenate(TEXT, TEXT) RETURNS TEXT
  STRICT IMMUTABLE PARALLEL SAFE ...

CREATE FUNCTION myrandom(INT, INT) RETURNS INT
  STRICT VOLATILE PARALLEL SAFE ...
```


Exercice : choisir les caractéristiques

null, volatilité

`int8mul`	multiplication d'entiers de 8 octets	
`NUM_NULLS`	compte le nombre d'arguments nuls	
`NOW`	donne l'heure de cette requête	
`pg_database_size`	taille d'une base de donnée	
`pg_sleep`	attend un nombre de secondes	
`COUNT`	aggrégation comptage de lignes	pas une fonction

Encadrement du code de la fonction

- différents langages possibles :
  SQL, PL/pgSQL, PL/Perl, PL/Python, PL/Tcl, PL/Java, C...
- fourniture du source
  chaîne de caractère SQL `'...'`, oblige à échapper les *quotes*
  séparateur générique `$marque...$` plus pratique !
- ou d'une librarie dynamique, précise librairie et symbole

Exemples d'encadrement


```
CREATE FUNCTION add(REAL, REAL) RETURNS REAL AS '
  SELECT $1 + $2;
' LANGUAGE sql IMMUTABLE STRICT;

CREATE FUNCTION add(REAL, REAL) RETURNS REAL AS $$
  BEGIN RETURN $1 + $2; END;
$$ LANGUAGE plpgsql IMMUTABLE STRICT;

CREATE FUNCTION add(REAL, REAL) RETURNS REAL AS $x$
  return $_[0] + $_[1];
$x$ LANGUAGE plperl IMMUTABLE STRICT;

CREATE FUNCTION add(REAL, REAL) RETURNS REAL AS $_$
  return args[0]+args[1]
$_$ LANGUAGE plpythonu IMMUTABLE STRICT;

CREATE FUNCTION add(REAL, REAL) RETURNS REAL
AS '$libdir/add', 'add_fun'
LANGUAGE C IMMUTABLE STRICT;
```


Attributs (colonnes) virtuels

- fonction sur le *type* de la relation
- appel style attribut : `relation.fonction`
- note : pourrait être fait avec une vue

```
CREATE TABLE People(firstname TEXT, lastname TEXT);

CREATE FUNCTION fullname(p People) RETURNS TEXT AS
$$ SELECT p.firstname || ' ' || p.lastname ; $$
LANGUAGE SQL IMMUTABLE STRICT;

SELECT ppl.fullname FROM People AS ppl;
-- Edgar Codd, Michael Stonebraker
```


Exercice : créer des attributs virtuels...


```
CREATE TABLE Personne
  (prénom TEXT, nom TEXT, né DATE, genre CHAR);

INSERT INTO Personne(nom, prénom, né, genre) VALUES
  ('Nette', 'Marion', '1970-10-14', 'F'),
  ('Mensoif', 'Gérard', '1970-03-20', 'M'),
  ('Némard', 'Jean', '2002-10-04', 'M'),
  ('Alise', 'Jeanne', '2000-04-16', 'F');

SELECT p.civilité, p.age FROM Personne AS p;
-- Mme Marion Nette, 49...
```

Déclaration de variables et du code en PL/pgSQL

- section initiale facultative DECLARE pour variables
  contraintes à la SQL : valeur par défaut, non nulle, constante
- code dans bloc BEGIN ...END;

```
DECLARE
  pi CONSTANT REAL DEFAULT 3.1415927;
  i INTEGER NOT NULL DEFAULT 3;
  r REAL DEFAULT NULL;
BEGIN
  r := 2.5;
  RETURN pi*(i+r);
END;
```


Affectation et opérations SQL

- expressions de SQL, comme un SELECT
  `i > 19 AND v IS NOT NULL`
- affectation directe à une variable

```
i := 3;
j := 2 * i + 1;
```

- opérations SQL directes après substitution des variables

```
s := 'ciel';
INSERT INTO def(mot,definition)
  VALUES (s,'en haut');
```

- ne pas faire grand chose...

```
-- commentaire pour mieux comprendre
NULL; -- pas d'opération
```


Récupération SELECT INTO STRICT var ...

- type spécial RECORD ou %ROWTYPE ou liste
- valeurs NULL si vide, un seul tuple si STRICT

```
DECLARE
  c RECORD; h pg_user%ROWTYPE;  name TEXT;  sysid INTEGER;
BEGIN
  SELECT * INTO STRICT c
    FROM pg_user WHERE usename='calvin';
  SELECT * INTO STRICT h
    FROM pg_user WHERE usename='hobbes';
  SELECT usename, usesysid INTO STRICT name, sysid
    FROM pg_user WHERE usename='postgres';
  RETURN c.name || ' ' || h.name || ' ' || name;
END;
```


Tester le fonctionnement d'un opération

- GET DIAGNOSTICS ... pour ROW_COUNT et RESULT_OID
- test si il s'est passé quelque chose : booléen FOUND

```
UPDATE comptes SET total = total+10.0;
GET DIAGNOSTICS ncomptes = ROW_COUNT;

SELECT total INTO montant
  FROM comptes WHERE id=12;
IF NOT FOUND THEN ...
```

Structures de contrôle : retour, condition

- retour obligatoire d'une fonction `RETURN`, retour de relations...
- condition `IF ... THEN ... ELSE ... END IF;`
  insertions possibles de `ELSEIF ... THEN ...`
  partie finale `ELSE` facultative

```
IF i=0 THEN
  RETURN 'zéro';
ELSEIF i>0 THEN
  RETURN 'positif';
ELSEIF i<0 THEN
  RETURN 'négatif';
ELSE
 RETURN 'nul';
END IF;
```


Structures de contrôle : boucles (1)

- boucle infinie `LOOP ... END LOOP;`
  sortie avec `EXIT`, éventuellement précise le niveau (bof)

```
i := 1;
LOOP
  i := i+1;
  EXIT WHEN i=10;
END LOOP;
```

- boucle conditionnelle `WHILE ... LOOP ... END LOOP;`

```
i := 1;
WHILE i<>10 LOOP
  i := i+1;
END LOOP;
```


Structures de contrôle : boucles (2)

- boucle sur entiers `FOR i IN x .. y LOOP ... END LOOP;`
  parcours inverse avec `REVERSE`

```
FOR i IN 1 .. n LOOP
  total := total + tab[i];
END LOOP;

FOR j IN REVERSE n .. 1 LOOP
  total := total + tab[j];
END LOOP;
```

- boucle sur une requête statique ou dynamique...

Structures de contrôle : exceptions

- bloc `BEGIN ... EXCEPTION ... END;` `try/except` Python
- nom des exceptions fixé, plus `others`
- envoie d'une exception avec `RAISE EXCEPTION ...`
  `RAISE NOTICE` produit un warning, utile pour debug !
  premier argument chaîne, remplacement des %

```
BEGIN
  i := j/k;
EXCEPTION
  WHEN division_by_zero THEN
    RAISE NOTICE 'boum i=% j=% k=%', i, j, k;
    RETURN NULL;
  -- sinon exception non attrapée
END;
```

Exécution de commandes SQL

- commandes directes INSERT DELETE UPDATE CREATE...
  positionnement de la variable booléenne spéciale FOUND...
- sélection avec un résultat SELECT INTO ...

```
SELECT usename INTO name
  FROM pg_user WHERE usesysid=id;
```

- sélection sans résultat PERFORM, pour effets de bords
  même syntaxe qu'un SELECT remplacé par PERFORM
  attention, pas de SELECT sans INTO direct

```
PERFORM change_passe(username, newpass);
```


Parcours FOR r IN query LOOP ... END LOOP;

- r de type RECORD générique ou ROWTYPE
- accès aux attributs r.prenom r.nom r.naissance

```
DECLARE
  r RECORD;
BEGIN
  FOR r IN
    SELECT * FROM pg_user WHERE usesysid>=100
  LOOP
    RAISE NOTICE 'user % num %', r.usename, r.usesysid;
  END LOOP;
  RETURN;
END;
```


Curseur CURSOR REFCURSOR

- nommage d'une requête SELECT en cours
- création et manipulation : OPEN FETCH MOVE CLOSE...
- peut être retourné d'une fonction !

```
DECLARE
  st RECORD;
  students REFCURSOR;
BEGIN
  OPEN students FOR SELECT * FROM Student;
  FOR st IN students LOOP
     -- ...
  END LOOP;
  CLOSE students;
END;
```


Interprétation de SQL avec EXECUTE

- requête inconnue lors de la définition de la fonction
- échappement identificateurs/litéraux quote_ident/literal
- fonction FORMAT avec %I (ident), %L (litéral), %s (string)

```
EXECUTE
  FORMAT('SELECT COUNT(*) FROM %I' , tn)
  INTO STRICT compte;
EXECUTE
  FORMAT('SELECT * FROM %I WHERE %I=%L', t, a, v)
  INTO r;
```

Parcours avec EXECUTE

- FOR r IN EXECUTE 'query' LOOP ... END LOOP;
- r de type RECORD ou ROW...
- aussi EXECUTE FORMAT ...

```
FOR r IN EXECUTE
  FORMAT('SELECT montant FROM %I', nom_table)
LOOP
  n := n + 1;
  somme := somme + r.montant;
END LOOP;
```


Retour d'un tuple, type composite


```
CREATE TYPE infos AS (nom TEXT, age INTEGER);
CREATE OR REPLACE FUNCTION suzy()
RETURNS infos AS $$
DECLARE
  r infos;
BEGIN
  r.nom := 'Suzy' ; r.age := 6;
  RETURN r;
END;
$$ LANGUAGE plpgsql;
```

nom	*age*
Suzy	6

Retour d'une relation

- retour d'un ensemble de quelque chose SETOF INTEGER
- tuples avec RETURN NEXT i, fin de la relation avec RETURN

```
CREATE OR REPLACE FUNCTION
un_a_n(n INTEGER) RETURNS SETOF INTEGER
AS $$
DECLARE
  i INTEGER;
BEGIN
  FOR i IN 1 .. n LOOP
    RETURN NEXT i;
  END LOOP;
  RETURN; -- fin de la relation
END;
$$ LANGUAGE plpgsql;
```


Utilisation comme une relation

mais l'optimiseur ne sait pas trop à quoi s'en tenir ?

```
SELECT * FROM un_a_n(3);
```

un_a_n
1
2
3

```
SELECT * FROM un_a_n(5);
```

un_a_n
1
2
3
4
5

Retour d'une relation de tuples


```
CREATE TYPE infos AS (nom TEXT, age INTEGER);

CREATE OR REPLACE FUNCTION nom_age()
RETURNS SETOF infos AS $$
DECLARE r infos;
BEGIN
  r.nom := 'Calvin'; r.age := 6; RETURN NEXT r;
  r.nom := 'Hobbes'; r.age := 3; RETURN NEXT r;
  RETURN; -- fin des sorties
END;
$$ LANGUAGE plpgsql;

SELECT * FROM nom_age();
```

nom	*age*
Calvin	6
Hobbes	3

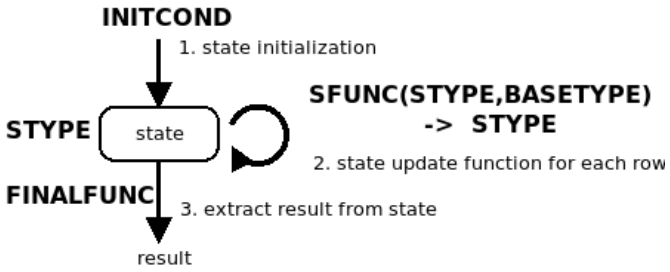
Définition d'un type CREATE TYPE ... AS ...

- type composite, structure de donnée *record*
  déjà utilisé par les relations
  `CREATE TYPE infos AS (prenom TEXT, nom TEXT);`
- ou nouveau type de plein droit avec fonctions en C
  input/output conversion de/vers chaîne de caractères
  receive/send conversion de/vers représentation binaire externe
  mais aussi ordre, indexation, analyse...

Fonctionnement d'une aggrégation

Transitions

état interne initialisé
fonctions de mises à jour et d'extraction finale



Définition d'une aggrégation

CREATE AGGREGATE ... (

BASETYPE = type de donnée concerné
SFUNC = fonction d'état : état, élément → état
STYPE = type de l'état stocké pendant le calcul
FINALFUNC = calcul le résultat final : état → élément
si non définie, retourne l'état
INITCOND = condition initiale de l'état (sinon NULL)
et d'autres pour aggrégats flottants *(moving aggregates)*
ou optimisation avec index, aggrégats dépendant d'un ordre...

Réimplementation simplifiée de COUNT


```
CREATE OR REPLACE FUNCTION fcompte(s INTEGER, x ANYELEMENT)
RETURNS INTEGER IMMUTABLE CALLED ON NULL INPUT AS $$
BEGIN
  IF x IS NULL THEN RETURN s;
  ELSE RETURN s+1; END IF;
END;
$$ LANGUAGE plpgsql;

CREATE AGGREGATE compte(
  BASETYPE=ANYELEMENT,
  SFUNC=fcompte,
  STYPE=INTEGER,
  INITCOND='0'
);

SELECT compte(titre) FROM oeuvre;
```


Aggrégat Moyenne


```
-- aggregate internal state
CREATE TYPE MoyStat AS (n INTEGER, s FLOAT);

-- state update function
CREATE OR REPLACE FUNCTION moyupd(state MoyStat, val FLOAT) RETURNS MoyStat
IMMUTABLE STRICT AS $$
BEGIN
  state.n := state.n + 1;  state.s := state.s + val;
  RETURN state;
END; $$ LANGUAGE plpgsql;

-- result extraction
CREATE OR REPLACE FUNCTION moyfin(state MoyStat) RETURNS FLOAT
IMMUTABLE STRICT AS $$ BEGIN RETURN state.s/state.n; END; $$ LANGUAGE plpgsql;

-- aggregate declaration
DROP AGGREGATE IF EXISTS moyenne(FLOAT);
CREATE AGGREGATE moyenne(
  BASETYPE = FLOAT, STYPE = MoyStat, SFUNC = moyupd,
  INITCOND = '(0, 0.0)', FINALFUNC = moyfin
);
```


Aggrégat Prems – *première valeur non nulle*


```
CREATE FUNCTION prems_sfunc(s ANYELEMENT, a ANYELEMENT)
RETURNS ANYELEMENT IMMUTABLE AS $$
  SELECT COALESCE(s, a);
$$ LANGUAGE SQL;

CREATE AGGREGATE PREMS(ANYELEMENT) (
  STYPE = ANYELEMENT,
  SFUNC = prems_sfunc
);

SELECT PREMS(titre) FROM oeuvre;
SELECT titre FROM oeuvre LIMIT 1;
```


Définition d'un opérateur CREATE OPERATOR ...

- opérateurs composés de séquences de caractères (qq exceptions)
  `+ - * / < > = ~ ! | ?...`
- nombreuses propriétés, associativité non modifiable

```
CREATE FUNCTION addtext(a TEXT, b TEXT)
RETURNS INTEGER IMMUTABLE STRICT AS $$
  BEGIN RETURN LENGTH(a) + LENGTH(b); END;
$$ LANGUAGE plpgsql;

CREATE OPERATOR +
( PROCEDURE = addtext, LEFTARG = TEXT, RIGHTARG = TEXT,
  COMMUTATOR = + -- + commute avec lui-même
);

SELECT 'hello' + 'world'; -- 10
```

Exercice : définir un opérateur contient


```
-- l'opérateur #@ teste la
-- présence d'une chaîne dans un texte
-- en ignorant la casse, avec LOWER et STRPOS
SELECT 'ob' #@ 'HoBBes'; -- TRUE
SELECT 'su' #@ 'Calvin'; -- FALSE
```


Définition d'un cast CREATE CAST ...

traduction entre deux types, fonction à fournir
3 niveaux explicite, AS ASSIGNMENT, auto AS IMPLICIT
différent de CONVERSION pour les encodages de caractères !

```
CREATE FUNCTION time2int(t TIMESTAMPTZ)
RETURNS INTEGER IMMUTABLE STRICT AS $$
  BEGIN RETURN EXTRACT(YEAR FROM t); END;
$$ LANGUAGE plpgsql;

CREATE CAST (TIMESTAMPTZ AS INTEGER)
  WITH FUNCTION time2int(TIMESTAMPTZ);

SELECT CAST(NOW() AS INTEGER);
SELECT NOW()::INTEGER;
```


Impact des cast implicites

- recherche automatique des opérateurs avec casts...

```
SELECT 2 * CURRENT_DATE; -- error

CREATE FUNCTION date2int(d DATE)
RETURNS INTEGER IMMUTABLE STRICT AS $$
  BEGIN RETURN EXTRACT(YEAR FROM d); END;
$$ LANGUAGE plpgsql;

CREATE CAST (DATE AS INTEGER)
WITH FUNCTION date2int(DATE)
AS IMPLICIT;

SELECT 2 * CURRENT_DATE; -- ok
```


Trigger sur DML : CREATE TRIGGER ...

Fonctions automatiques

- fonction spéciale invoquée automatiquement
- quand : **avant**, **après**, **à la place** (vues) — BEFORE, AFTER, INSTEAD OF
- évènements — INSERT UPDATE DELETE TRUNCATE
- granularité tuple vs table — FOR EACH ROW/STATEMENT
- éventuellement : condition, références — WHERE ...

```
CREATE TRIGGER coucou_compte
  BEFORE INSERT OR UPDATE OR DELETE
  ON compte FOR EACH ROW
  EXECUTE PROCEDURE trace('salut');
```

Programmation d'une trigger en PL/pgSQL

Trigger PL/pgSQL

- pas d'arguments, retour type `TRIGGER`
- accès aux informations par variables spéciales définies selon cas
  - `OLD` ancien tuple (`RECORD`) pour `UPDATE DELETE`
  - `NEW` nouveau tuple pour `INSERT UPDATE`
  - `TG_NAME TG_RELID TG_SCHEMA_NAME TG_TABLE_NAME` nom de la trigger et relation
  - `TG_WHEN TG_LEVEL` `BEFORE`/`AFTER`/..., `ROW`/`STATEMENT`
  - `TG_OP` opération en cours `INSERT UPDATE`...
  - `TG_NARGS TG_ARGV` nombre et tableau des arguments texte
- retour du nouveau tuple (modifiable si `BEFORE`), de l'ancien ou levée d'une exception pour refuser...

Trigger sur DDL : `CREATE EVENT TRIGGER ...`

- spécificité PostgreSQL – non standard
- event : `ddl_command_start`/`end` `sql_drop` `table_rewrite`
- tag : presque toutes les opérations sur objets locaux
  matrice de déclanchement selon les événements
  mais pas sur objects globaux `DATABASE` `ROLE` `TABLESPACE`
  ni sur `EVENT TRIGGER` (risque de blocage !)
- programmation en C ou PL/pgSQL, type spécial `event_trigger`

```
CREATE EVENT TRIGGER table_event ON ddl_command_start
WHEN TAG IN ('CREATE TABLE', 'ALTER TABLE', 'DROP TABLE')
EXECUTE PROCEDURE do_something();
```


Exemple de trigger DML


```
CREATE TABLE entiers(i INTEGER);

CREATE FUNCTION trace() RETURNS TRIGGER AS $$
BEGIN
  RAISE NOTICE '% % % on %.%',
    TG_OP, TG_WHEN, TG_LEVEL, TG_SCHEMA_NAME, TG_TABLE_NAME;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trace_entiers BEFORE INSERT OR UPDATE
ON entiers FOR EACH STATEMENT EXECUTE PROCEDURE trace();

INSERT entiers(i) VALUES(1);
-- NOTICE:  INSERT BEFORE STATEMENT on PUBLIC.entiers
-- INSERT 0 1
```


Exemple de trigger DDL


```
CREATE FUNCTION do_something()
RETURNS event_trigger AS $$
BEGIN
  RAISE NOTICE 'event %: %', TG_EVENT, TG_TAG;
END;
$$ LANGUAGE plpgsql;

CREATE TABLE foo(stuff TEXT);
-- NOTICE:  event ddl_command_start: CREATE TABLE
DROP TABLE foo;
-- NOTICE:  event ddl_command_start: DROP TABLE
```

Utilisations possibles des triggers

- vérifications de contraintes (clefs étrangères, . . .)
- mise à jour automatique d'attributs
  date de dernière mise à jour du tuple. . .
- contrôle de cohérence avec requêtes (attention aux récursions ?!)
  blocage de tuples sous certaines conditions
- blocage de certaines opérations DDL
- duplication logique asynchrone de tables (Slony-I)
  évènements envoyés vers une autre base
- possible d'activer/désactiver un `TRIGGER`

```
ALTER TABLE oeuvre
  DISABLE TRIGGER oeuvre_check_titre;
```


Conclusion sur PL/pgSQL

- un **langage** pas extraordinaire mais très utile !
  bien intégré car basé sur SQL
- déboguage peu commode
  erreurs de syntaxe tardives pour ;
  utiliser des `RAISE NOTICE ...` pour tracer
  éventuellement dans des expressions. . .
- performance : programmation directe en C
  plus délicate, *untrusted* : peut planter la base
- tout n'est pas possible
  librairies dynamiques additionnelles en C ?
  développement dans un langage plus généraliste ?