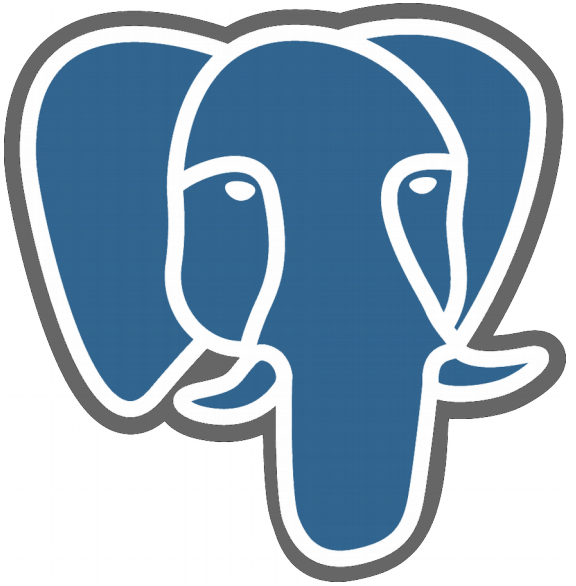
PL/pgSQL

INSTRUCTOR:

Emilio Pérez <info@todopostgresql.com>



PostgreSQL Procedural Language

PostgreSQL permite escribir funciones definidas por el usuario en multitud de lenguajes de programación.

- PL/pgSQL
- PL/Tcl
- PL/Perl
- PL/Python
- PL/Java
- PL/Ruby

PostgreSQL Procedural Language

- Pl/pgSQL puede ser usado para:
  - Crear funciones y triggers
  - Añadir estructuras de control al lenguaje SQL
  - Optimiza computaciones complejas
  - Hereda todos los tipos definidos por el usuario, las funciones y los operadores
  - Es facil de usar

¿Como trabaja?

- PL/pgSQL define estructuras de bloques para escribir código
- Las funciones pueden ser compiladas y almacenadas dentro del servidor ofreciendo un mejor rendimiento

Estructura de bloque

PL/pgSQL es un lenguaje de estructura de bloques

- DECKARE (opcional)
  - Variables, cursores, excepciones definidas por el usuario
- BEGIN
  - Sentencias SQL
  - Sentencias PL/pgSQL

Estructura de bloque

- EXCEPTION (opcional)
  - Actiones a realizar cuando ocurra un error.
- END;

Estructura de bloque

- Cada declaración y cada sentencia de un bloque es terminado por un punto y coma.

- Un bloque que se encuentra dentro de otro bloque debe tener un punto y coma después del END

- El END final que concluye el cuerpo de la función no necesita terminar por punto y coma.

Estructura de bloque

- Todas las palabras claves e identificadores pueden escribirse en mayúsculas y en minúsculas.

- Los identificadores se convierten implícitamente a minúsculas a menos que lo pongas entre doble comillado.

Estructura de bloque

Disponemos de dos tipos de comentarios

- Comentarios de una sóla línea:

```
-- Esta línea está comentada
```

- Comentarios multilíneas

```
/* Este comentario puede tener varias líneas */
```

Declaración de variables

Variables

- Almacenan datos temporales.
- Pueden ser reusadas dentro del bloque.
- Declarada e inicializada en la sección de declaraciones.
- Usada y asignada en la sección de ejecución.

Declaración de variables

Variables

- Pasado como parámetros a subprogramas.
- Usadas para capturar la salida de un subprograma.
- Puede ser cualquier tipo de datos válidos de SQL.
- Puede contener las cláusulas DEFAULT o CONSTANT.

Sintaxis

Nombre [CONSTANT] tipo

[NOT NULL] [{DEFAULT | := } expresión]

Ejemplos

```
id_usuario integer;
cantidad numeric(5);
direccion varchar;
mitupla nombretabla%ROWTYPE;
micampo nombretabla.nombrecolumna%TYPE;
registro RECORD;
edad integer DEFAULT 21;
```

Asignando valores a variables

Asignaciones

- identificador := expresion;
- id_usuario := 20;
- Impuestos := subtotal * 0.21;

SELECT INTO

- SELECT INTO variable select_expresion FROM ... ;
- SELECT INTO miregistro * FROM empleados WHERE nombre = minombre;

Variables locales

FOUND

- Variable local booleana.
- Es inicializada a false en cada llamada de PL/pgSQL.
- Es inicializado por las siguientes tipos de sentencias:
  - Las sentencias SELECT INTO asignan verdadero a FOUND si este retorna una tupla y falso si no retorna ninguna.

Variables locales

FOUND

- Las sentencias PERFORM asignan verdadero a FOUND si este produce una tupla y falso si no la produce.
- UPDATE, INSERT y DELETE asignan verdadero a FOUND si al menos una tupla es afectada y falso si no ha sido afectada ninguna.

Variables locales

FOUND

- Una sentencia FETCH asigna verdadero a FOUND si retorna una tupla y falso si no retorna ninguna.
- Una sentencia FOR asigna verdadero a FOUND si itera una o más veces, si no, asigna falso.

Escribiendo sentencias

- Las sentencias ejecutables son escritas en bloques BEGIN .. END
- Los bloques pueden ser anidados.
- Las sentencias dentro de un bloque pueden ocupar varias líneas
- Todas las sentencias deben finalizar con ;

Declarando parámetros

- Los parámetros pasados a una función son nombrados con identificadores $1, $2, etc
- De manera opcional se le puede poner un alias.
- Se puede usar tanto el alias como el identificador numérico.
- CREATE FUNCTION aplicar_impuestos (subtotal real)

Ejemplo

```
CREATE OR REPLACE FUNCTION sumar(i numeric,
j numeric)
RETURNS numeric AS $$
begin
RETURN i+j;
end;
$$ LANGUAGE plpgsql;
```

Ejemplo

```
postgres=# SELECT sumar(5,6);
 sumar
-------
    11
(1 row)
```

Ejemplo

```
CREATE OR REPLACE FUNCTION restar(i numeric,
j numeric)
RETURNS void AS $$
DECLARE d numeric;
begin
d:= i-j;
raise notice 'La resta es: %',d;
RETURN;
end;
$$ LANGUAGE plpgsql;
```

Ejemplo

```
postgres=# SELECT restar(8,6);
NOTICE:  La resta es: 2
 restar
--------


(1 row)
```

Estructuras de control

Utilizadas para cambiar el flujo de control en los bloques plpgsql.

Disponemos de tres tipos de estructuras de control:

- IF
- CASE
- LOOP

IF

Equivalente a las estructuras IF en otros lenguajes de programación.

Sintaxis:

```
IF expresion-booleana THEN
   sentencia
[ELSIF expresion-booleana THEN sentencias ]
[ ELSE sentencias ]
END IF;
```

Ejemplo

```
CREATE OR REPLACE FUNCTION cliente(id_cliente
numeric)
RETURNS void AS $$
DECLARE tupla RECORD;
```

Ejemplo

```
begin
SELECT INTO tupla* FROM clientes WHERE
id=id_cliente;
if FOUND then
raise notice 'Nombre de cliente: %',tupla.nombre;
raise notice 'Número de cuenta: %',tupla.cuenta;
else
raise notice 'El cliente no existe';
end if;
```

Ejemplo

```
RETURN;
end;
$$ LANGUAGE plpgsql;
```

Ejemplo

```
postgres=# select * from clientes;
 id |   nombre   | cuenta
----+------------+--------
  1 | cliente1   |   1001
  2 | cliente2   |   1002
  3 | cliente3   |   1003
  4 |            |
(4 rows)
```

Ejemplo

```
postgres=# SELECT cliente(1);
NOTICE:  Nombre de cliente: cliente1
NOTICE:  Número de cuenta: 1001
 cliente
---------


(1 row)
```

Ejemplo

```
postgres=# SELECT cliente(4);
NOTICE:  Nombre de cliente: <NULL>
NOTICE:  Número de cuenta: <NULL>
 cliente
---------


(1 row)
```

Ejemplo

```
postgres=# SELECT cliente(5);
NOTICE:  El cliente no existe
 cliente
---------


(1 row)
```

CASE

Retorna un resultado basado en una o más alternativas.

El valor del selector determina cual es el resultado que es retornado.

CASE

Sintaxis:

CASE expresión-busqueda

WHEN expresion [, expresion […]] THEN

sentencia

[WHEN expresion [, expresion […]] THEN

sentencias …]

[ELSE sentencias]

END CASE;

EJEMPLO

```
CREATE OR REPLACE FUNCTION temperatura(tem
numeric) RETURNS varchar AS $$

DECLARE sms varchar;
```

EJEMPLO

```
BEGIN

CASE WHEN tem < 0 THEN
sms := 'Hielo';
WHEN tem between 0 and 10 THEN
sms := 'Frio';
WHEN tem > 10 THEN
sms := 'NORMAL';

ELSE
sms := 'Sin determinar';

END CASE;
```

EJEMPLO

```
RETURN sms;
END;
$$ LANGUAGE plpgsql;
```

EJEMPLO

```
postgres=# SELECT temperatura(-1);
 temperatura
-------------
 Hielo
(1 row)

postgres=# SELECT temperatura(0);
 temperatura
-------------
 Frio
(1 row)
```

```
postgres=# SELECT temperatura(6);
 temperatura
-------------
 Frio
(1 row)

postgres=# SELECT temperatura(10);
 temperatura
-------------
 Frio
(1 row)
```

```
postgres=# SELECT temperatura(15);
 temperatura
-------------
 NORMAL
(1 row)
```

LOOP

Utilizado en ejecuciones que deben ser repetidas hasta que se cumpla una condición de salida.

Hay tres tipos básicos de loop:

- Loop Básico
- WHILE
- FOR

LOOP BÁSICO

Es utilizado cuando se realizan acciones repetidas sin condiciones generales.

Debe tener un EXIT para salir del bucle

LOOP BÁSICO

Sintaxis:

LOOP

  Sentencias

Sintaxis de salida

EXIT [etiqueta] [WHEN expresion-booleana];

END LOOP;

EJEMPLO

```
CREATE OR REPLACE FUNCTION incremental()
RETURNS void AS $$
DECLARE rec numeric;
```

EJEMPLO

```
begin
rec:=0;
loop
raise notice '%',rec;
rec:=rec+1;
EXIT when rec>10;
end loop;
```

EJEMPLO

```
RETURN;
end;
$$ LANGUAGE plpgsql;
```

WHILE

Acciones repetitivas basadas en una condición.

La condición es evaluada al comienzo de cada iteración.

Si la condición proporciona NULL, se saldrá del bucle.

WHILE

Sintaxis:

WHILE expresion-booleana LOOP

Sentencias

END LOOP;

EJEMPLO

```
CREATE OR REPLACE FUNCTION aumentar()
RETURNS void AS $$
DECLARE rec numeric;
```

EJEMPLO

```
begin
rec:=0;
while rec<=10 loop
raise notice '%',rec;
rec:=rec+1;
end loop;
```

EJEMPLO

```
RETURN;
end;
$$ LANGUAGE plpgsql;
```

FOR

Los bucles FOR iteran basados en una cantidad.

Tienen la misma estructura que un bucle básico.

Antes de la palabra reservada LOOP se encuentra una sentencia de control para asignar el número de iteraciones.

FOR

Sintaxis:

FOR nombre IN [REVERSE] expresion .. expresion [BY expresion] LOOP

Sentencias

END LOOP;

EJEMPLO

```
CREATE OR REPLACE FUNCTION multi()
RETURNS void AS $$
DECLARE rec numeric;
```

EJEMPLO

```
begin
for rec in 0..10 by 2 loop
raise notice '%',rec;
end loop;
RETURN;
end;
$$ LANGUAGE plpgsql;
```

INTERCEPTANDO ERRORES

Los errores de sintaxis se capturan en tiempo de compilación.

Plpgsql puede causar algunos errores en tiempo de ejecución.

El tratamiento de estos errores se realizan en el bloque EXCEPTION.

INTERCEPTANDO ERRORES

Sintaxis:

[DECLARE

declarations]

BEGIN

statements

INTERCEPTANDO ERRORES

Sintaxis:

EXCEPTION

WHEN condition [OR condition ...] THEN

handler_statements

[WHEN condition [OR condition ...] THEN

handler_statements

...]

END;

INTERCEPTANDO ERRORES

La palabra reservada EXCEPTION inicia la sección de manejo de excepciones.

Nos permite capturar múltiples excepciones.

Sólo se procesa un capturador antes de dejar el bloque.

INTERCEPTANDO ERRORES

WHEN OTHERS es la última sentencia y puede capturar todos los tipos de excepciones.

El bloque EXCEPTION se encuentra al final del bloque BEGIN --- END.

EJEMPLO

```
CREATE OR REPLACE FUNCTION div
    (a numeric, b numeric)
RETURNS numeric AS $$
DECLARE result numeric;
```

EJEMPLO

```
begin
result=a/b;
EXCEPTION
When others then
raise notice 'Valor incorrecto para el segundo parámetro.
Debe ser un valor distinto de cero.';
```

EJEMPLO

```
return result;
end;
$$ LANGUAGE plpgsql;
```

CURSORES

Declaración de cursores

- DECLARE curs1 refcursor;
- DECLARE curs2 CURSOR FOR SELECT * FROM tenk1;
- DECLARE curs3 CURSOR (key integer) IS SELECT *
  FROM tenk1 WHERE unique1 = key;

CURSORES

Abriendo cursores

- OPEN FOR query
- OPEN unbound_cursor FOR query;
- OPEN curs1 FOR SELECT * FROM foo
  WHERE key = mykey;

CURSORES

Abriendo cursores

- OPEN FOR EXECUTE
- OPEN unbound_cursor FOR EXECUTE query_string;
- OPEN curs1 FOR EXECUTE 'SELECT * FROM'
  || quote_ident ($1);

CURSORES

Abriendo cursores enlazados

- OPEN bound_cursor [(argument_values)];
  abriendo cursor OPEN FOR query

- DECLARE

curs2 CURSOR FOR SELECT * FROM tenk1;

curs3 CURSOR (key integer) IS SELECT * FROM tenk1
WHERE unique1 = key;

CURSORES

Abriendo cursores enlazados

```
OPEN curs2;
OPEN curs3(42);
```

CURSORES

FETCH

```
FETCH cursor INTO target;
```

FETCH proporciona la siguiente tupla del cursor y la almacena en target, el cual puede ser una variable de tipo tupla, un registro o una lista de variables separada por comas.

CURSORES

FETCH

```
FETCH cursor INTO target;
```

FETCH curs1 INTO rowvar;

FETCH curs2 INTO foo, bar, baz;

La variable FOUND es chequeada a verdadero si se devuelve una tupla.

CURSORES

CLOSE

CLOSE cierra el cursor;

Se utiliza para liberar recursos antes de finalizar la transacción o liberar variables de cursor para que pueda ser abierta de nuevo.

```
CLOSE curs1;
```

CURSORES

```
CREATE OR REPLACE FUNCTION cliente_cursor()
RETURNS void AS $$
DECLARE
cur CURSOR for SELECT*FROM clientes;
rec RECORD;
```

CURSORES

```
begin
open cur;
fetch cur into rec;
while found loop
raise notice 'Nombre de cliente: %', rec.nombre;
raise notice 'Número de cuenta: %', rec.cuenta;
fetch cur into rec;
end loop;
```

CURSORES

```
raise notice 'No hay más clientes';
close cur;
RETURN;
end; $$ language plpgsql;
```

TRIGGERS

Se crean con el comando CREATE FUNTION.

Varias variables especiales son creadas de manera automática a nivel de bloque:

- NEW: tipo de dato registro. Contiene los valores de la nueva tupla en operaciones de INSERT/UPDATE.

- OLD: tipo de dato registro. Contiene los valores anteiores de la tupla en operaciones de UPDATE/DELETE.

TRIGGERS

Varias variables especiales son creadas de manera automática a nivel de bloque:

- TG_NAME: tipo de dato nombre. Esta variable contiene el nombre del trigger que se encuentra ejecutando.

- TG_WHEN: tipo de dato texto. Cadena de BEFORE o AFTER dependiendo de la definición del disparador.

TRIGGERS

Varias variables especiales son creadas de manera automática a nivel de bloque:

- TG_LEVEL: tipo de dato texto. Cadena de ROW o STATEMENT dependiendo de la definición del disparador.

- TG_OP: tipo de dato texto. Cadena de INSERT, UPDATE o DELETE indicando qué operación activó el disparador.

TRIGGERS

Varias variables especiales son creadas de manera automática a nivel de bloque:

- TG_RELNAME: tipo de dato nombre. El nombre de la tabla que causó la invocación del disparador.

- TG_NARGS: tipo de dato entero. El número de argumentos dado al procedimiento en la sentencia CREATE TRIGGER.

TRIGGERS

Nota: Una función de trigger debe devolver NULL o un valor de registro / fila que tenga exactamente la estructura de la tabla por la que se disparó el trigger.

El valor de retorno de un trigger de sentencias de nivel de declaración BEFORE o AFTER o un trigger de sentencia de nivel de fila AFTER siempre se ignora; También puede ser nulo.

TRIGGERS

Sin embargo, cualquiera de estos tipos de desencadenadores pueden abortar toda la operación al generar un error.

EJEMPLO

CREATE table ciudad(ciudad_id numeric, poblacion numeric);

CREATE table nacimiento(reg_id numeric, name varchar, ciudad_id numeric);

insert into ciudad values(1,0),(2,0);

select * from ciudad;

EJEMPLO

```
CREATE OR REPLACE FUNCTION trg_ins() RETURNS
trigger AS $$

begin

update ciudad set poblacion=poblacion+1 where
ciudad_id=NEW.ciudad_id;

return null;

end; $$ language plpgsql;
```

EJEMPLO

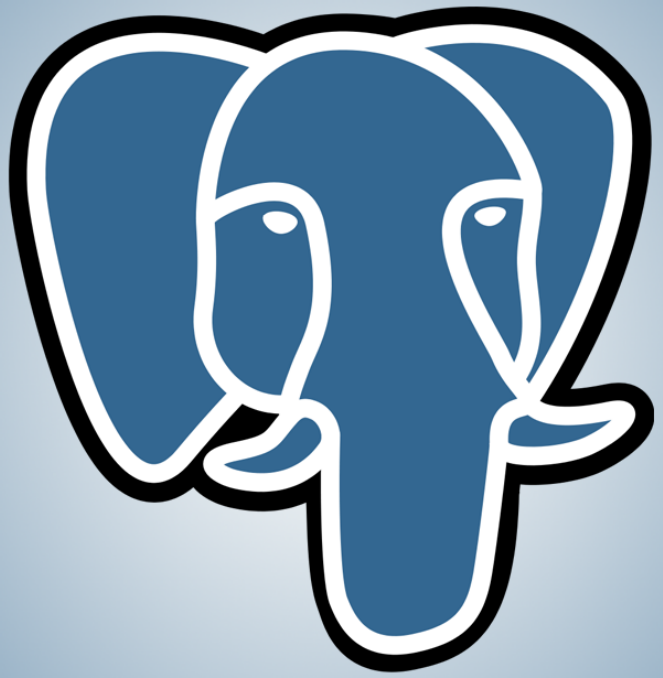
```
create trigger trg_ins_nacimiento after insert on
nacimiento

for each row execute procedure trg_ins();


insert into nacimiento values(101,'Raj',2);

select*from nacimiento;

select*from ciudad;
```



todopostgresql.com