Programación en PL/PGSQL

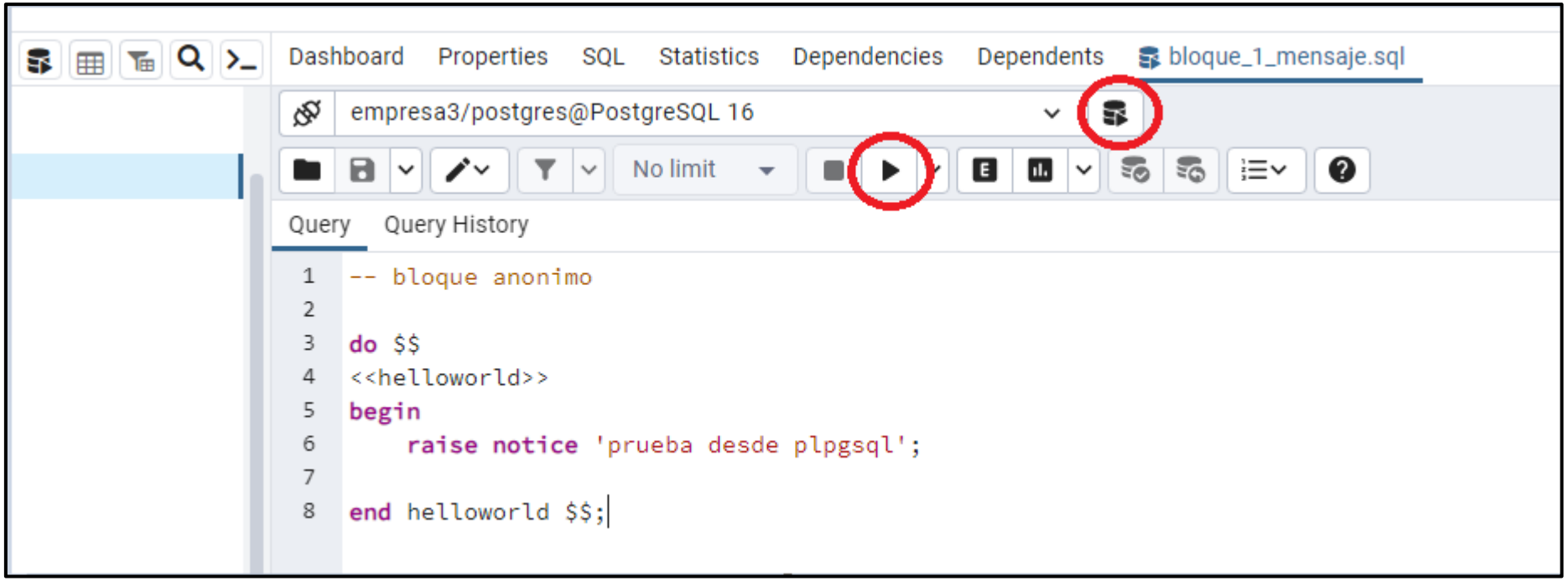
Antonio Espín Herranz

Contenidos

- Comentarios
- Estructura
- Sentencias
- Variables
- Control de flujo
- Retornar valores
- Mensajes

Query tool

- Esta opción de **pgAdmin** se puede utilizar para ejecutar consultas o bloques de código **PL/PGSQL**:



Comentarios

- Puede ser comentario en línea para comentar sólo una línea:

-- comentario de una línea

- Comentario en bloque para comentar varias líneas:

/* Comentario
En varias líneas */

Estructura

- Podemos tener bloques de código de PL/PGSQL dentro de **funciones** y **procedimientos** almacenados o dentro de **bloques anónimos**:

```
-- Comentarios en línea: Emite un mensaje por la consola de pgAdmin 4
do $$
<<helloworld>>
begin
        raise notice 'prueba desde plpgsql';
end helloworld $$;
```

Las etiquetas en los bloques anónimos son opcionales.

Ejemplo

- También lo podemos ver así, sin etiquetar:

```
do $$
begin
        raise notice 'prueba desde plpgsql';
end $$;
```

Estructura

- La estructura general de un bloque de PL/pgSQL:

```
[ <<label>> ]
[ declare declarations ]
begin
statements; ...
end [ label ];
```

- Las variables irán definidas dentro del apartado declare .

Estructura

- Los bloques anónimos se ejecutan desde la herramienta query tool integrada dentro de pgAdmin.

- A estos bloques también se les puede asignar un nombre y se almacenan dentro del servidor.

- En este caso tenemos dos tipos:
  - Funciones almacenadas: realizan un cálculo y lo devuelven. Se pueden ejecutar dentro de una sentencia SQL o se llamarán desde otro bloque pl/pgsql.
  - Procedimientos almacenados: ejecutar una serie de tareas y se llamarán desde otro bloque, pero nunca desde una sentencia SQL.
  - Ambos podrán recibir parámetros.

Estructuras

- Los bloques de pl/pgsql se pueden anidar, dentro del begin – end del bloque externo se vuelve a definir otro bloque.

```
do $$
<<bloque_externo>>
begin
        raise notice 'dentro del bloque externo';

        <<bloque_interno>>
        begin
                raise notice 'dentro del bloque interno';
        end bloque_interno;

end bloque_externo $$;
```

Estructura

- https://www.postgresqltutorial.com/postgresql-plpgsql/plpgsql-block-structure/

- https://www.postgresql.org/docs/current/plpgsql-overview.html#PLPGSQL-ADVANTAGES

Sentencias

- Las sentencias se colocan entre begin y end y terminan con un ;
- Se van a poder mezclar sentencias de pl/pgsql con instrucciones sql.
- El código se puede escribir en minúsculas o mayúsculas.
- Con los bloques de código vamos a poder agrupar varias sentencias SQL incluyendo cálculos intermedios en vez de ir ejecutando cada consulta por separado y esperar el resultado (como nos ocurre cuando estamos interactuando con el servidor mediante SQL).

Variables

- Las variables irán dentro del apartado **declare**.

- Indicando un identificador, el tipo (que será un tipo de postgre) y también se puede dar un valor inicial.

- Dentro de los identificadores no utilizar los guiones medios, si permite los subrayados.

Variables

- Cuando tenemos dos o más bloques anidados podemos declarar variables a varios niveles (dentro de cada bloque), el bloque interno tendrá acceso a las variables del bloque externo.

- Si las variables se llamaran igual, en el bloque interno tendremos acceso a la más próxima. Es decir, la variable del bloque interno ocultará a la variable del bloque externo.

Variables

- Declare
  - cantidad integer := 30;
  - valor integer;

- Se pueden inicializar o no.

- Para asignar un valor a una variable fuera del apartado declare (entre begin-end) lo haremos con :=
- valor:= 50;

Variables

- Para imprimir el valor de una variable:
- **raise notice ‘El valor de cantidad es %’, cantidad;**
- El **%** indica el punto de inserción de esa variable en el mensaje.
- Tendremos que poner tantos % como distintas variables tengamos.

Variables

```
Do $$
Declare
        entero integer:=100;
        real float:= 45.77;
        texto varchar(20):='mensaje’;
Begin
    raise notice 'impresion de variables: % % %', entero, real, texto;
End $$;
```

Variables

- Carga de una variable con el resultado de una consulta:
- Select función_agregado **into variable** from tabla

- Por ejemplo:

```
Declare
    numClientes integer;
Begin
    Select count(*) into numClientes from tbclientes;
    Raise notice ‘el número de clientes es: %’, numClientes;
End;
```

Variables

- Se pueden asignar alias a variables.
- Declare
  - newName ALIAS FOR oldname;
-
- En SQL era aconsejable cuando definíamos campos calculados o realizamos consultas con varias tablas (ayudan a simplificar las sentencias SQL).

Variables

- Se pueden definir tipos de variables que sean del mismo tipo que una columna de una tabla o de una tabla completa (con todos los campos).

- **%TYPE**
  - Proporciona el tipo de datos de una tabla o de una columna de la tabla.

- **%ROWTYPE**
  - Define una variable con una fila de la tabla.

Variables

- %**type**
- Es muy cómodo para definir una variable del tipo de una columna.

- Si cambia el tipo de la columna no tenemos que tocar el script siempre y cuando las operaciones realizadas con esa variable sean compatibles con el nuevo tipo de la columna.

- Declare
  - Nombre_var nombre_tabla.nombre_col%**type**;

Variables

- %**rowtype**
- Para capturar un registro completo / fila de una tabla:
  - Declare
    - Nombre_var nombre_tabla%**rowtype**;
  - Begin
    - Select * into nombre_var from nombre_tabla where cond
  - End

- Si no ponemos condición coge el primero de la tabla.
- La variable se puede imprimir entera, muestra la fila entre paréntesis.
- O podemos acceder a los campos con el punto: nombre_var.nombre_col

Variables

- Tipo **Record**

- Similar al rowtype, pero no tiene una estructura predefinida como ocurre rowtype que lo asociamos a una tabla concreta.

- Tomará la estructura del registro a partir de la consulta ejecutada.

- Se puede asignar primero un registro de la tabla 1 y después otro de la tabla2. La estructura del record irá cambiando según se ejecuta una consulta u otra.

- Declare
  - Registro **record**;

Control de Flujo

- Condicionales:
  - **If**: Permite anidamientos, else, etc.
  - **Case**: Evaluación múltiple. Mas cómodo de leer y mejora el rendimiento.

- Bucles:
  - **Loop**
  - **For**
  - **While**

Sentencia IF

- IF boolean-expression THEN
- statements
- [ELSIF boolean-expression THEN
- statements
- [ELSIF boolean-expression THEN
- statements
- ...
-]
-]
- [ELSE
- statements]
- END IF;

Sentencia IF

If cond then procesamiento1;

[Elsif cond then procesamiento2;]

Else

Procesamiento3;

End if;

- Permite **elsif** anidados y prescindir de la parte Else.
- Los operadores para montar las condiciones son los mismos que en SQL:

  =, >, <, !, >=, <=, is null, is not null, between, like, and, or.

Ejemplo

```
If varnumcli = 10 then
    Update clientes set nombre='Juan' where numcli = varnumcli;
    Commit;
Else
    Rollback;
End if;
```

Sentencia Case

- Nos permite la evaluación múltiple.
- Se puede realizar la evaluación por valor o por condición.
- Nos permite añadir una claúsula else por si no se cumple ninguno de los casos.
- Sintaxis: **Evaluando por valor.**
  - [<<etiqueta>>]
  - **Case** elemento
    - When valor1 then instrucciones1;
    - When valor2 then instrucciones2;
    - [Else instrucciones;]
  - **End Case** [etiqueta];

Sentencia Case

- Sintaxis **evaluando por condición**:
  - [<<Etiqueta>>]
  - **Case**
    - When condición1 then instrucciones1;
    - When condición2 then instrucciones2;
    - [Else instrucciones];
  - **End case** [etiqueta];

Ejemplos

- **Evaluación por valor:**

```
Do $$
declare
 provincia integer:=32;
 nombre VARCHAR(40);

 begin
  case provincia
    when 32 then nombre:='Orense';
    when 36 then nombre:='Pontevedra';
    else nombre:= 'fuera de la comunidad';
  end case;

 end $$;
```

- **Evaluación por condición:**

```
Do $$
declare
   provincia integer:=32;
   comunidad VARCHAR(40);
 begin
  case
   when provincia in (02,13,16,19,45) THEN
     comunidad:='Castilla la mancha';
   when provincia in (04,11,14,18,21,23,29,41) then
     comunidad:='Andalucía';
   else
     comunidad:='Otra comunidad';
   end case;
 end  $$;
```

Sentencia Case

- En caso de que no se añada la sentencia else, y no se cumpla ningún caso saltará un error.
- **ERROR: caso no encontrado**

- **La sentencia case se puede utilizar dentro del SQL.**

Bucle Loop

- Se trata de la variedad más simple de la sentencia LOOP, y se corresponde con el bucle básico (o infinito), el cual incluye una secuencia de sentencias entre las palabras claves LOOP y END LOOP.
- Ejemplo:
  - *LOOP*
    - *secuencia_de_sentencias;*
  - *END LOOP;*

Bucle Loop

- Tenemos dos formas de salir del bucle:
  - EXIT y EXIT WHEN.

- La sentencia **EXIT** provoca la **salida** de un bucle de forma **incondicional**. Cuando se encuentra un EXIT, el bucle acaba inmediatamente y el control pasa a la siguiente instrucción que esté fuera del bucle.

- Ejemplo:
  - LOOP
    - …
    - IF limite_credito < 3 THEN
    - …
    - EXIT; -- Fuerza la salida inmediata…
    - END IF;
  - END LOOP;

Bucle Loop

- La sentencia **EXIT-WHEN**, nos va a permitir salir de un bucle de forma condicional. Cuando PL/PGSQL encuentra una sentencia de este tipo, la condición del WHEN será evaluada… en caso de devolver TRUE, se provocará la salida del bucle… en caso contrario, se continuará la iteración.

- Ejemplo:

```
LOOP
    …
        EXIT WHEN contador>100;
END LOOP
```

Bucles con etiquetas

- Al igual que los bloques de PL/SQL, los bucles pueden ser etiquetados. La etiqueta es un identificador no declarado que se escribe entre los símbolos << y >>; deben aparecer al principio de las sentencias LOOP.

```
<<mi_loop>>
LOOP
        secuencia_de_sentencias;
END LOOP mi_loop;
```

- Con las etiquetas podemos forzar la salida de bucles internos, salir hasta el bucle más externo.

Bucles con etiquetas

```
<<exterior>>
LOOP
  …
  LOOP
        …
        EXIT exterior WHEN …
        -- Sale de los dos bucles LOOP…
  END LOOP;
  …
END LOOP exterior; -- exterior aquí no es necesario.
```

Sentencia Continue

- Esta sentencia permite interrumpir la iteración en curso y pasar directamente a la siguiente.
- Sintaxis:

  Continue [label] [when condición]

```
LOOP
   -- some computations
   EXIT WHEN count > 100;
   CONTINUE WHEN count < 50;
   -- some computations for count IN [50 .. 100]
END LOOP;
```

Bucle For

- Permite ejecutar las instrucciones dentro del bucle haciendo variar un índice.
- Las instrucciones se ejecutan tantas veces como cambia el índice de valor.
- El índice se puede utilizar dentro de las instrucciones, pero siempre en modo lectura.

Bucle For

- Sintaxis:

  <<etiqueta>>

  For indice in [reverse] exp1…exp2 by expresión loop

  Instrucciones

  …

  End loop [etiqueta]

  - El índice se declara de forma implícita.
  - Exp1, exp2 son constantes, expresiones o variables.
  - Si no se utiliza reverse el índice avanza de exp1 a exp2 de uno en uno.
  - Si se utiliza reverse, el índice varía de exp2 a exp1 con incrementos de -1.
  - Con by expresión podemos incrementar de dos en dos u otro valor.

Bucle For

- Ejemplo:

```
For n in 100..110 loop
    Insert into clientes (numcli) values (n);
End loop;
Commit;
```

- El bucle se ejecuta 10 veces, y se ejecuta el sql.

Bucle for a través de una query

- El bucle for también se puede utilizar para recorrer el resultado de una consulta.

[<<label>>]
FOR target IN **query** LOOP
    statements
END LOOP [label];

- Target puede ser una variable de tipo **record**, rowtype o una lista de valores escalares que almacenarán el resultado de la consulta.
- En cada iteración el bucle sirve una fila del resultado de la consulta.

Bucle For

- El bucle for también se puede utilizar con una consulta para recorrer mediante un cursor el resultado de la query:

```
Declare
    Resultado record; --El registro toma las columnas de la consulta.

Begin
    For resultado in (select * from clientes) loop
        Raise info 'texto %', resultado.item;
    End loop;

end
```

Bucle for

- También se puede pasar instrucciones SQL mediante un string y se le pueden pasar parámetros cuando se van a ejecutar:

```
Declare
    Varsql text;
Begin
    Varsql:= 'select … from …where campo=$1 and campo2=$2';
    For row in execute varsql [using param1, …]   -- Los parámetros son opcionales!
    Loop
        código
    End loop;
End
```

Execute sql

- También se puede ejecutar una consulta SQL a través de un string (de una variable) sin tener que estar en un bucle.

- Execute ‘create table tabla(….)’;
- Execute ‘drop table tabla’;

- A veces tendremos que concatenar alguna variable con un sql. Si la variable necesita comillas porque es un texto utilizaremos la función: quoute_literal para colocar las comillas.
  - **Select quote_literal (44); → ‘44’**

Execute sql

- Dentro del SQL que pasamos a la instrucción execute no podemos utilizar into variable.

- No está implementado, y si lo utilizamos obtendremos un error.

Bucle While

- La evaluación del bucle se realiza al inicio.
- Si la condición es true el bucle se ejecuta.
- Sintaxis:
  - <<etiqueta>>
  - While condición loop
    - Instrucciones;
    - …
  - End loop [etiqueta];
- En la condición se pueden utilizar todos los operadores: <,>,=,<>, AND, OR, LIKE, …

Ejemplo

```
x:= 1;
while x <= 10 loop
    insert into clientes (numcli) values (x);
    x := x + 1;
end loop;
commit;
```

- Crea clientes del 1 al 10, ambos incluidos.

Bucles

- En los bucles: loop y while podemos salir o continuar:

```
Begin
    …
    Exit [when cond.]
    …
End

Begin
    …
    continue [when cond.]
    …
end
```

Bucle foreach a través de un array

- El bucle **foreach** permite interactuar con los elementos de un array.

[<<label>>]

FOREACH target [SLICE number] IN ARRAY expression LOOP

    statements

END LOOP [label];

Ejemplo

```
do $$
declare
        numeros int[];
        numeros2 int[]:=array[1,2,3,4];  -- Inicializar el array
        item integer;
begin
for i in 0..10 loop
        numeros[i]:=i*100;
end loop;

foreach item in array numeros  loop
        raise notice 'item: %', item;
end loop;
end $$;
```

Retornar valores

- Se utiliza la palabra **return,** pero dentro del contexto de una función almacenada.

- La variable o expresión que se devuelve debe coincidir con el tipo que indica que devuelve la función.

Cursores

```
do $$
declare
        categoria record;
        cur_cat cursor(idcat integer) for select * from tbcategorias where id > idcat;

begin
        open cur_cat(5);

        loop
                fetch cur_cat into categoria;
                exit when not found;
                        raise info 'cat: %', categoria;
        end loop;
        close cur_cat;

end $$;
```

Excepciones

- Errores que se producen en tiempo de ejecución y abortan la ejecución del script mostrando el error en la consola.

- Además de poder capturar los errores en postgre también vamos a poder lanzar errores ante una situación que nosotros consideremos que se trata de un error.

Estructura completa

```
<<label>>
declare
begin

statements;

exception
   when condition [or condition...] then
      handle_exception;
   [when condition [or condition...] then
      handle_exception;]
   [when others then
      handle_other_exceptions; ]
end;
```

Orden de ejecución

- Cómo funciona.
  - Primero, cuando ocurre un error entre begin y exception, PL/pgSQL detiene la ejecución y pasa el control a la lista de excepciones.
  - En segundo lugar, PL/pgSQL busca el primero condition que coincida con el error que se produce.
  - En tercer lugar, si hay una coincidencia, handle_exception se ejecutarán las declaraciones correspondientes. PL/pgSQL pasa el control a la declaración después de la end palabra clave.
  - Finalmente, si no se encuentra ninguna coincidencia, el error se propaga y puede ser detectado por la cláusula exception del bloque adjunto. En caso de que no haya un bloque adjunto con la cláusula exception, PL/pgSQL abortará el procesamiento.

Excepciones

- Postgre mantiene una lista de códigos de error:
- Cada error tiene asociado un código y mensaje asociado.
- La lista se encuentra:
  - https://www.postgresql.org/docs/current/errcodes-appendix.html

Excepciones

- Estructura del bloque plpgsql cuando tenemos que capturar excepciones.
  - Do $$
  - Declare
  - …
  - Begin

  - Exceptions
    - When nombre_excepción then
      - Normalmente se imprime un mensaje con el error
      - Se pueden indicar otras sentencias
      - Los mensajes se lanzan con raise notice .. U otro nivel de error.
  - End $$;

Excepciones

- Por ejemplo, hacemos una consulta y con la constante found podemos saber si se han obtenido resultados o no.

- En caso de que no podemos optar por lanzar una excepción.

```
Select …
If not found
    Raise exception “mensaje asociado a la excepción: %’, variable;
End if;
```

Excepciones

- Cuando realizamos una consulta y estamos recogiendo datos en una variable, puede pasar que no haya datos o que se devuelvan varias filas.

- Si estamos utilizando el modo estricto postgre informará del error lanzando una excepción que podemos capturar:

- Select campo into **strict** variable from tabla where condición;

Ejemplo

```
BEGIN
 SELECT * INTO STRICT myrec FROM emp WHERE empname = myname;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            RAISE EXCEPTION 'employee % not found', myname;
        WHEN TOO_MANY_ROWS THEN
            RAISE EXCEPTION 'employee % not unique', myname;
END;
```

Capturar el error

- Podemos capturar información del error que ocurre:

```
DECLARE
  text_var1 text;
  text_var2 text;
  text_var3 text;

BEGIN
  -- Alguna instrucción causa un error!
  ...
EXCEPTION WHEN OTHERS THEN
  GET STACKED DIAGNOSTICS text_var1 = MESSAGE_TEXT,
                          text_var2 = PG_EXCEPTION_DETAIL,
                          text_var3 = PG_EXCEPTION_HINT;
END;
```

Mensajes

- Los mensajes se pueden lanzar con **raise**.

- Esta palabra clave también se utiliza para lanzar errores.

- Un mensaje:
  - **Raise** notice ‘el mensaje es %’, mensaje;
  - La variable mensaje se incrusta en el %

Mensajes

- El nivel de los mensajes puede ser:
- **Info**
- **Notice**
- **Warning**

- En la consola se mostrará el nivel del mensaje.
- Normalmente estos 3 niveles están activos en la configuración por defecto, en:
  - **client_min_message** y **log_min_messages**

Mensajes

- Los niveles de **LOG** y **DEBUG** normalmente no se muestran al cliente bajo las configuraciones predeterminadas de **client_min_messages** y **log_min_messages**.

- Se muestran igual que los anteriores:
- Raise **debug** ‘mensaje de depuración’;

Mensajes

- El nivel de mensajes que queremos mostrar se puede cambiar con la propiedad client_min_messages

- Set client_min_messages = debug;

Enlaces

- https://www.postgresql.org/docs/current/plpgsql.html