**Postgresql – Funções**

**(FUNCTIONS ou PROCEDURES)**

Cada banco de dados implementa funções de uma forma um pouco diferente uns dos outros. Para os exemplos aqui listados, se utilizará as funções no SGBD PostgreSQL.

**Funções (ou procedimentos) são trechos de código que ficam armazenados dentro do banco de dados.**

Da mesma forma que existem funções prontas no banco de dados (exemplo: SUM, AVG, MAX…), pode-se criar várias funções.

Para o Postgresql, o mesmo suporta muitos tipos de linguagem para a criação de funções dentro do banco de dados.

CREATE [ OR REPLACE ] FUNCTION

...] ] )

RETURNS *tipo\_retornado*

{ LANGUAGE *nome\_da\_linguagem*

| IMMUTABLE | STABLE | VOLATILE

| CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT

| [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER

| AS '*definição*'

| AS '*arquivo\_objeto*

} ...

[ WITH ( *atributo [,* ...] ) ]

**Exemplo Função Simples SQL:**

CREATE OR REPLACE FUNCTION somar\_inteiros(integer, integer)

RETURNS integer AS 'select $1 + $2;'

LANGUAGE SQL IMMUTABLE

RETURNS NULL ON NULL INPUT;

SELECT somar\_inteiros(2,3);

**Exemplo Função Simples SQL com parâmetro TABELA:**

drop table empregados

CREATE TEMP TABLE empregados (

nome text,

salario numeric,

idade integer

);

INSERT INTO empregados VALUES('lucas',2200,21);

INSERT INTO empregados VALUES('marcelo',4200,30);

INSERT INTO empregados VALUES('felipe',3200,19);

INSERT INTO empregados VALUES('Danton',3500,24);

INSERT INTO empregados VALUES('felipe',3200,22);

INSERT INTO empregados VALUES('Jean',3700,18);

INSERT INTO empregados VALUES('Eduardo',3100,21);

INSERT INTO empregados VALUES('Jean',3300,19);

INSERT INTO empregados VALUES('Leonardo',3600,19);

INSERT INTO empregados VALUES('Luis',3500,19);

select \* from empregados

drop function dobrar\_salario(empregados)

CREATE or REPLACE FUNCTION dobrar\_salario(empregados) RETURNS numeric AS $$

SELECT $1.salario \* 2 AS salario\_dobrado;

$$ LANGUAGE SQL;

SELECT nome, salario, dobrar\_salario(empregados)

FROM empregados;

**Algumas vezes é prático gerar o valor do argumento composto em tempo de execução. Isto pode ser feito através da construção ROW.**

SELECT nome, dobrar\_salario(ROW(nome, salario\*1.1, idade)) AS sonho

FROM empregados;

**---FUNÇÕES com utilização da Linguagem plpgsql;**

A sintaxe para a criação de uma nova função ou procedimento é (de forma simplificada):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | CREATE OR REPLACE FUNCTION nome\_da\_funcao ( parametros )      RETURNS tipo\_retorno AS $$  DECLARE        --declaracao de variaveis  BEGIN        --conteudo da funcao  END;  $$ LANGUAGE linguagem; |

**-- Usando comando IF**

**CREATE OR REPLACE**

**FUNCTION operacao(numeric, char(1),numeric)**

**RETURNS varchar(100) AS $$**

**declare retorno varchar(100);**

**declare resultado numeric(7,2);**

**declare obs varchar (20)= ' ';**

**BEGIN**

**if $2 = '+' then**

**resultado= $1+$3;**

**else if $2 = '-' then**

**resultado= $1-$3;**

**else if $2 = '\*' then**

**resultado= $1\*$3;**

**else if $2 = '/' then**

**resultado= $1/$3;**

**else**

**Resultado = 0;**

**retorno = 'operador inválido';**

**return retorno;**

**end if;**

**end if;**

**end if;**

**end if;**

**retorno = $1 || $2 ||$3 || ' = ' || resultado || obs ;**

**RETURN retorno ;**

**END;**

**$$ LANGUAGE plpgsql;**

**select operacao(3,'\*',4);**

**select operacao(3,'+',4);**

**select operacao(3,'-',4);**

**Select operacao(15,'/',2);**

**select operacao(12,'$',3)**

**--usando comando FOR**

**CREATE OR REPLACE**

**FUNCTION elevar(numeric, char(1),numeric)**

**RETURNS varchar(100) AS $$**

**declare i integer;**

**declare retorno varchar(100);**

**declare resultado numeric(7,2);**

**declare obs varchar (20)= ' ';**

**BEGIN**

**resultado = 1;**

**if $2 = '^' then**

**FOR i IN 1..$3 LOOP**

**RESULTADO = resultado \* $1;**

**END LOOP;**

**retorno = $1 || $2 ||$3 || ' = ' || resultado || obs ;**

**RETURN retorno ;**

**else**

**Resultado = 0;**

**retorno = 'operador inválido';**

**return retorno;**

**end if;**

**END;**

**$$ LANGUAGE plpgsql;**

**SELECT ELEVAR (2,'^',3)**

**SELECT ELEVAR (2,'^',4)**

**SELECT ELEVAR (2,'^',5)**

**SELECT ELEVAR (4,'^',5)**

**--Usando comando CASE**

**CREATE or replace FUNCTION calcular(NUMERIC, char(1), NUMERIC) RETURNS NUMERIC AS $$**

**DECLARE**

**resultado NUMERIC(10,2);**

**BEGIN**

**CASE**

**WHEN $2 = '+' THEN resultado = $1 + $3;**

**WHEN $2 = '-' THEN resultado = $1 - $3;**

**WHEN $2 = '/' THEN resultado = $1 / $3;**

**WHEN $2 = '\*' OR $2 = 'x' OR $2 = '.' THEN resultado = $1 \* $3;**

**ELSE RAISE NOTICE 'Operador Inválido!';**

**END CASE;**

**RETURN resultado;**

**END;**

**$$ LANGUAGE plpgsql;**

**SELECT CALCULAR(2,'.',3)**

**SELECT CALCULAR(2,'/',3)**

**SELECT CALCULAR(2,'^',3)**

**-- usando comando WHILE**

**create or replace function exponencial(numeric,numeric)**

**RETURNS varchar(100) as $$**

**DECLARE igual char(1);**

**DECLARE retorno varchar(100);**

**DECLARE resultado numeric;**

**DECLARE cont integer;**

**DECLARE elevado varchar(10);**

**begin**

**igual = '=';**

**elevado ='^';**

**cont=1;**

**resultado = $1;**

**while cont < $2 loop**

**resultado = resultado \* $1;**

**cont = cont +1 ;**

**end loop;**

**retorno = $1 || elevado || $2 || igual || resultado ;**

**return retorno;**

**end;**

**$$ LANGUAGE plpgsql;**

**select exponencial(2,3)**

**select exponencial(2.5,3)**

**select elevar (2,'^',3)**

**Exemplo prático:**

Criar a tabela no banco de dados:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | CREATE TABLE usuario (  id integer NOT NULL,  nm\_login character varying,  ds\_senha character varying,  fg\_bloqueado boolean,  nu\_tentativa\_login integer,  CONSTRAINT pk\_usuario PRIMARY KEY (id)  ); |

Na tabela ‘usuario’, temos os seguintes dados:

INSERT INTO usuario values(1, 'hallan', 'hallan2011','false',0);

INSERT INTO usuario values(2, 'joao', '123456','false',0);

INSERT INTO usuario values(3, 'maria', 'abcd1234','false',2);

select \* from usuario;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ID | NM\_LOGIN | DS\_SENHA | FG\_BLOQUEADO | NU\_TENTATIVA\_LOGIN |
| 1 | Hallan | hallan2011 | False | 0 |
| 2 | João | 123456 | False | 0 |
| 3 | Maria | abcd1234 | False | 2 |

Um exemplo básico de uma função seria criar uma função onde se passa o login do usuário e a função retorna o ID do usuario.

Exemplo: Pode-se criar, então, a função da seguinte forma:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | CREATE OR REPLACE FUNCTION get\_id ( **varchar** ) RETURNS integer AS  $$  DECLARE          variavel\_id INTEGER;  BEGIN          SELECT INTO variavel\_id id FROM usuario WHERE nm\_login = $1;          RETURN variavel\_id;  END;  $$ LANGUAGE 'plpgsql'; |

A função acima (chamada **get\_id**) recebe como parâmetro um VARCHAR e retorna um INTEGER, e funciona da seguinte forma:

- declara uma variável chamada **variavel\_id**, do tipo INTEGER;

- faz um select na tabelas de usuário onde a coluna **nm\_login** é igual ao **varchar** recebido como parâmetro, e colocar o valor de **id** encontrado dentro da variável **variavel\_id** (através do comando **SELECT INTO**);

- retorna a **variavel\_id**.

O $1 representa o primeiro parâmetro recebido, o $2 o segundo, e assim por diante.

Agora, se executarmos a seguinte instrução SQL:

|  |  |
| --- | --- |
| 1 | SELECT get\_id( 'joao' ); |

O resultado do resultset:

|  |
| --- |
| GET\_ID |
| 2 |

**FUNÇÃO COM ATUALIZAÇÃO DE TABELA**

**EXEMPLO**

**REGRA:** se o usuário entrar com a senha errada três vezes seguidas, ele deve ser bloqueado.

A coluna ‘fg\_bloqueado’ mostra se o usuário está bloqueado ou não, e a coluna ‘nu\_tentativa\_login’ mostra quantas vezes seguidas este usuário inseriu sua senha errada.

Exemplo: se um usuário que tem o número de tentativas (coluna **nu\_tentativa\_login**) = 1 entrar com sua senha errada, o numero de tentativas deverá ser alterado para 2. Caso entre novamente com a senha errada, o número de tentativas deverá ser alterado para 3 **E** a coluna **fg\_bloqueado** deverá ser alterada para TRUE.

Caso um usuário com o número de tentativas = 2 tenha entrado com a senha correta, o valor do número de tentativas deverá voltar a ser 0.

Toda esta lógica pode ser implementada diretamente no banco de dados, através de uma função.

Seja a seguinte função:

CREATE OR REPLACE FUNCTION set\_tentativa\_login ( VARCHAR, VARCHAR )

RETURNS VOID AS

$$

DECLARE

registro RECORD;

tentativas INTEGER;

BEGIN

--raise notice 'teste';

SELECT INTO registro id, fg\_bloqueado, nu\_tentativa\_login FROM usuario

WHERE nm\_login = $1 AND ds\_senha = $2;

IF registro IS NULL -- Não encontrou Login nem senha correspondente

THEN

SELECT INTO tentativas nu\_tentativa\_login FROM usuario

WHERE nm\_login = $1;

tentativas := tentativas + 1;

IF tentativas > 2 -- verifica numero de tentativas

THEN

UPDATE usuario SET nu\_tentativa\_login = tentativas,

fg\_bloqueado = TRUE where nm\_login = $1;

RAISE notice 'Ultrapassou numero de tentativas válidas';

ELSE

UPDATE usuario SET nu\_tentativa\_login = tentativas

where nm\_login = $1;

END IF;

ELSE --Encontrou Login e senha correspondente

Raise notice 'Usuario e senha OK!'; -- Mensagem mostrada na area de mensagens

UPDATE usuario SET nu\_tentativa\_login = 0 where nm\_login = $1;

END IF;

END;

$$

LANGUAGE 'plpgsql';

É possível realizar estruturas de decisão e de repetição em funções de bancos de dados.

A função **set\_tentativa\_login** recebe dois parâmetros: um sendo o login **$1** e o outro a senha do usuário **$2**. Primeiro é verificado se existe um usuário com o login e senha informado. O resultado é colocado dentro de uma variável do tipo **RECORD**, que representa um registro (com várias colunas) de uma tabela.

Caso exista o usuário, seu número de tentativas é alterado para ZERO. Caso não exista, é efetuada outra consulta, desta vez somente através de seu **login**. O número de tentativas é armazenado na variável tentativas, e este valor é incrementado em 1. Além de atualizar o número de tentativas, é verificado se este valor é maior que 2. Caso seja, altera o valor da coluna **fg\_bloqueado** para TRUE.

Na prática:

Executando a seguinte instrução:

|  |  |
| --- | --- |
| 1 | SELECT set\_tentativa\_login( 'hallan', 'senha\_errada' ); |

A tabela de usuários ficará da seguinte forma:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ID | NM\_LOGIN | DS\_SENHA | FG\_BLOQUEADO | NU\_TENTATIVA\_LOGIN |
| 1 | Hallan | hallan2011 | False | 1 |
| 2 | João | 123456 | False | 0 |
| 3 | Maria | abcd1234 | False | 2 |

Como foi passada a senha errada, o número de tentativas foi incrementado em 1.

Caso seja executado a SQL:

|  |  |
| --- | --- |
| 1 | SELECT set\_tentativa\_login( 'maria', 'senha\_errada' ); |

A tabela de usuários ficará da seguinte forma:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ID | NM\_LOGIN | DS\_SENHA | FG\_BLOQUEADO | NU\_TENTATIVA\_LOGIN |
| 1 | Hallan | hallan2011 | False | 1 |
| 2 | João | 123456 | False | 0 |
| 3 | Maria | abcd1234 | True | 3 |

Além da senha, o usuário maria teve sua coluna **fg\_bloqueado** alterado para TRUE.

Ao executar:

|  |  |
| --- | --- |
| 1 | SELECT set\_tentativa\_login( 'hallan', 'hallan2011' ); |

A tabela de usuários ficará da seguinte forma:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ID | NM\_LOGIN | DS\_SENHA | FG\_BLOQUEADO | NU\_TENTATIVA\_LOGIN |
| 1 | Hallan | hallan2011 | false | 0 |
| 2 | João | 123456 | false | 0 |
| 3 | Maria | abcd1234 | true | 3 |

O valor da coluna **nu\_tentativa\_login** será alterado para ZERO, pois desta vez o valor do campo senha estava correto.

**FUNÇÃO com Correção para acesso**

CREATE OR REPLACE FUNCTION set\_verifica\_login ( VARCHAR, VARCHAR )

RETURNS VOID AS

$$

DECLARE

registro RECORD;

tentativas INTEGER;

BEGIN

SELECT INTO registro

id,

nm\_login,

ds\_senha,

fg\_bloqueado,

nu\_tentativa\_login FROM usuario

WHERE nm\_login = $1; -- Busca o registro no banco de dados

IF registro IS NOT NULL THEN -- Se o usuário for encontrado...

IF registro.fg\_bloqueado = FALSE THEN -- Se o usuário NÃO estiver bloqueado...

IF registro.ds\_senha = $2 THEN -- Se a senha estiver CORRETA...

Raise notice 'Usuario e senha OK!';

UPDATE usuario SET nu\_tentativa\_login = 0 WHERE nm\_login = $1; -- Zera o número de tentativas

ELSE -- Se a senha estiver INCORRETA...

tentativas := registro.nu\_tentativa\_login + 1;

IF tentativas > 2 THEN -- Se o limite de tentativas for atingido...

UPDATE usuario SET nu\_tentativa\_login = tentativas,

fg\_bloqueado = TRUE WHERE nm\_login = $1;

RAISE notice 'Usuário ou senha inválidos. Você atingiu o limite de tentativas!';

ELSE -- Se o limite de tentativas NÃO for atingido...

UPDATE usuario SET nu\_tentativa\_login = tentativas WHERE nm\_login = $1;

RAISE NOTICE 'Usuario ou senha não encontrados!';

END IF;

END IF;

ELSE -- Se o Usuário estiver bloqueado

tentativas := registro.nu\_tentativa\_login + 1;

UPDATE usuario SET nu\_tentativa\_login = tentativas WHERE nm\_login = $1;

RAISE NOTICE 'Usuário bloqueado. Procure um administrador do sistema!';

END IF;

ELSE -- Se o usuário não for encontrado...

RAISE NOTICE 'Usuario ou senha não encontrados!';

END IF;

END;

$$

LANGUAGE 'plpgsql';

SELECT set\_verifica\_login( 'hallan', 'hallan2011' );

**RETORNAR MAIS DE UM RESULTADO COM UMA FUNÇÃO,**

Criar a tabela que será a base das consultas:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | **CREATE TABLE pessoa\_fisica (**  **id\_pessoa SERIAL,**      nome VARCHAR(80),      sobrenome VARCHAR(200),      sexo CHAR(1),      cpf CHAR(11),      PRIMARY KEY(id\_pessoa)  ); |

**INSERT INTO PESSOA\_FISICA (NOME, SOBRENOME, SEXO,CPF) VALUES('JOAO','SILVA',1,'55566677788');**

**INSERT INTO PESSOA\_FISICA (NOME, SOBRENOME, SEXO,CPF) VALUES('MARIA','SOUZA',0,'11122233344');**

**1) TIPO RETORNO RECORD**

Neste tipo de FUNÇÂO é necessário especificar as colunas que se irá resgatar quando chamar a função

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | CREATE FUNCTION **get\_pessoas**() RETURNS **SETOF** RECORD AS $$  BEGIN      RETURN QUERY SELECT id\_pessoa, nome, sobrenome, sexo, cpf FROM pessoa\_fisica;      RETURN;  END;  $$ LANGUAGE 'plpgsql';  **--Chamada da Função**  SELECT \* FROM **get\_pessoas**() AS (  id\_pessoa INTEGER,  nome VARCHAR(80),  sobrenome VARCHAR(200),  sexo CHAR(1),  cpf CHAR(11)) |

Quando se define o tipo de retorno da função se determina dois parâmetros, **SETOF** que indica que a função irá retornar um conjunto de itens, ao invés de um único item e **RECORD** que está dizendo que o retorno será um conjunto de resultados.

 Utiliza-se a instrução **RETURN QUERY** e logo após um  **SELECT** tradicional para “capturar” os dados.

A instrução **RETURN** (sozinha, na linha 4) é que irá realizar o retorno do conjunto.

Chama-se a função, como se fosse uma tabela.

O **SELECT** faz um “alias” especificando quais campos vieram na query realizada. Deve-se colocar todos os campos, conforme a query que foi realizada na função.

**2) RETORNANDO OS CAMPOS DE UMA DADA TABELA**

Quando o retorno da função forem oriundos de apenas uma tabela, pode-se setar o tipo de retorno como sendo exatamente o da tabela trabalhada:

Definindo  **pessoa\_fisica**, como o tipo de retorno:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | CREATE FUNCTION blog\_get\_pessoas2() RETURNS SETOF pessoa\_fisica AS $$  BEGIN      RETURN QUERY SELECT \* FROM pessoa\_fisica      RETURN;  END;  $$ LANGUAGE 'plpgsql'    SELECT \* FROM blog\_get\_pessoas2(); |

Neste exemplo o retorno possível é de todos os campos da tabela  **pessoa\_fisica**,

Ao chamar a função pode-se selecionar os campos que interessam .

**OBS.: não se precisa especificar quais campos eventualmente viriam na query**

**3) TIPO DE RETORNO RETURNS TABLE**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  **7**  8  9  10  11  12 | CREATE FUNCTION blog\_get\_pessoas3() RETURNS TABLE (  id\_pessoa INT,                             nome VARCHAR,                             sobrenome VARCHAR,                             sexo CHAR,                             cpf CHAR) AS $$  BEGIN  **RETURN QUERY SELECT \* FROM pessoa\_fisica;**      RETURN;  END;  $$ LANGUAGE 'plpgsql'    SELECT \* FROM blog\_get\_pessoas3(); |

Declara-se os atributos, que serão os “nomes de coluna” quando se chama a função. É muito útil quando retorna dados de mais de uma tabela, dessa maneira pode-se expressar no corpo da função, explicitamente, quais serão as tuplas no retorno.

**OBS:** Na linha **7** é selecionando todos os campos, deve-se cuidar para que todos os campos coincidam com os determinados no tipo de retorno.

**4) CRIANDO UM TIPO ESPECÍFICO DE RETORNO, O CREATE TYPE**

Retorno da função “personalizado”.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7** | **CREATE TYPE type\_pessoa\_fisica AS (**  **id\_pessoa INT,**  **nome VARCHAR,**  **sobrenome VARCHAR,**  **sexo CHAR,**  **cpf CHAR**  **);** |

A instrução **CREATE TYPE** é bem semelhante a instrução para criação de tabelas, com ela é possivel criar um “mix” de duas tabelas por exemplo:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | CREATE FUNCTION blog\_get\_pessoas4() RETURNS SETOF type\_pessoa\_fisica AS $$  DECLARE      dados\_pessoa type\_pessoa\_fisica;  BEGIN      FOR dados\_pessoa IN SELECT id\_pessoa, nome, sobrenome, sexo, cpf  FROM pessoa\_fisica LOOP          RETURN NEXT dados\_pessoa;      END LOOP;      RETURN;  END;  $$ LANGUAGE 'plpgsql'    SELECT \* FROM blog\_get\_pessoas4(); |

1. O tipo de retorno da função, o **type\_pessoa\_fisica** que já especifica quais os campos que serão retornados,
2. O **DECLARE** define uma variável chamada **dados\_pessoa** que irá conter o conteudo da query, que tem o**mesmo tipo de retorno da função**, exatamente porque é esta variável que retornará.
3. **FOR-IN-SELECT** serve para abastecer a variável com os registros da tabela, que é um looping que a cada iteração “coloca uma linha na variável”,
4. **RETURN NEXT** é que “abastece” nossa variável, e apesar de ser um **“RETURN”** não encerra a execução do código naquele instante,
5. A instrução **RETURN** após o fim do loop (**END LOOP**), esta sim irá terminar a função.
6. Na **linha 12** tem a chamada da função, bem mais amigável que em comparação com o experimento 1, e é (quase) exatamente como a chamada de uma tabela, que se pode introduzir as clausulas **WHERE** e **LIMIT** por exemplo.

**5) CRIANDO VARIÁVEIS DENTRO DE BLOCOS (BEGIN – END)**

Deve-se ter cuidado com o escopo das variáveis dentro das funções e blocos de comandos.

**CREATE or replace FUNCTION somefunc() RETURNS integer AS $$**

**<< outerblock >>**

**DECLARE**

**quantity integer := 30;**

**BEGIN**

**RAISE NOTICE 'Quantity here inicio is %', quantity; -- Prints 30**

**quantity := 50;**

**--**

**-- Create a subblock**

**--**

**DECLARE**

**quantity integer := 80;**

**BEGIN**

**RAISE NOTICE 'Quantity here is %', quantity; -- Prints 80**

**RAISE NOTICE 'Outer quantity here is %', outerblock.quantity; -- Prints 50**

**END;**

**RAISE NOTICE 'Quantity here is %', quantity; -- Prints 50**

**RETURN quantity;**

**END;**

**$$ LANGUAGE plpgsql;**

**Referências:**

* [PostgreSQL Documentation – Create Function](http://www.postgresql.org/docs/8.3/static/sql-createfunction.html)
* [WikiBooks – Funções em PL/pgSQL](http://pt.wikibooks.org/wiki/PostgreSQL_Pr%C3%A1tico/Fun%C3%A7%C3%B5es_Definidas_pelo_Usu%C3%A1rio_e_Triggers/PlpgSQL)
* [PostgreSQL Brasil – Retornando registros de consultas genéricas com RETURN QUERY](http://postgresqlbr.blogspot.com/2008/12/retornando-registros-de-consultas.html)
* <http://blog.hallanmedeiros.com/docencia/banco-de-dados-tutorial/postgresql-funcoes>
* <http://pt.wikibooks.org/wiki/PostgreSQL_Pr%C3%A1tico/Fun%C3%A7%C3%B5es_Definidas_pelo_Usu%C3%A1rio_e_Triggers/PlpgSQL>