**Docente: Fábio Giulian Marques** prof.fabiomarques@gmail.com 

# 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.

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

```
1  CREATE OR REPLACE FUNCTION nome_da_funcao ( parametros )
2    RETURNS tipo_retorno AS $$
3  DECLARE
4      --declaracao de variaveis
5  BEGIN
6      --conteudo da funcao
7  END;
   $$ LANGUAGE linguagem;
```

**Exemplo prático:**

Criar a tabela no banco de dados:

```
1   CREATE TABLE usuario (
2   id integer NOT NULL,
3   nm_login character varying,
4   ds_senha character varying,
5   fg_bloqueado boolean,
6   nu_tentativa_login integer,
7   CONSTRAINT pk_usuario PRIMARY KEY (id)
8   );
```

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 |

**Docente: Fábio Giulian Marques** prof.fabiomarques@gmail.com **Módulo/Semestre: 3º SEMESTRE**

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   CREATE OR REPLACE FUNCTION get_id ( varchar ) RETURNS integer AS
2   $$
3   DECLARE
4       variavel_id INTEGER;
5   BEGIN
6       SELECT INTO variavel_id id FROM usuario WHERE nm_login = $1;
7       RETURN variavel_id;
8   END;
9   $$ 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.

**Docente: Fábio Giulian Marques** prof.fabiomarques@gmail.com 

Seja a seguinte função:

```
CREATE OR REPLACE FUNCTION set_tentativa_login ( VARCHAR, VARCHAR )
   RETURNS VOID AS
$$
DECLARE
      registro RECORD;
      tentativas INTEGER;
BEGIN
      SELECT INTO registro id, fg_bloqueado, nu_tentativa_login FROM usuario
                  WHERE nm_login = $1 AND ds_senha = $2;
      IF registro IS NULL -- Não encontrou Login nem senha correspondente
      THEN
         SELECT INTO tentativas nu_tentativa_login FROM usuario
                  WHERE nm_login = $1;
         tentativas := tentativas + 1;
         IF tentativas > 2 -- verifica numero de tentativas
         THEN
            UPDATE usuario SET nu_tentativa_login = tentativas,
                        fg_bloqueado = TRUE where nm_login = $1;
         ELSE
            UPDATE usuario SET nu_tentativa_login = tentativas
                        where nm_login = $1;
         END IF;
      ELSE  --Encontrou Login e senha correspondente
         UPDATE usuario SET nu_tentativa_login = 0 where nm_login = $1;
      END IF;
END;
$$
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:

```
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.

**Docente: Fábio Giulian Marques** prof.fabiomarques@gmail.com 

## RETORNAR MAIS DE UM RESULTADO COM UMA FUNÇÃO,

Criar a tabela que será a base das consultas:

```
1    CREATE TABLE pessoa_fisica (
2       id_pessoa SERIAL,
3       nome VARCHAR(80),
4       sobrenome VARCHAR(200),
5       sexo CHAR(1),
6       cpf CHAR(11),
7       PRIMARY KEY(id_pessoa)
8    );
```

```
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    CREATE FUNCTION get_pessoas() RETURNS SETOF RECORD AS $$
2    BEGIN
3       RETURN QUERY SELECT id_pessoa, nome, sobrenome, sexo, cpf FROM pessoa_fisica;
4       RETURN;
5    END;
6    $$ LANGUAGE 'plpgsql';
7            --Chamada da Função
8    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.

**Docente: Fábio Giulian Marques** prof.fabiomarques@gmail.com 

## 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  CREATE FUNCTION blog_get_pessoas2() RETURNS SETOF pessoa_fisica
2  AS $$
3  BEGIN
4     RETURN QUERY SELECT * FROM pessoa_fisica
5     RETURN;
6  END;
7  $$ LANGUAGE 'plpgsql'
8
   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 RETUNS TABLE

```
   CREATE FUNCTION blog_get_pessoas3() RETURNS TABLE (
1                    id_pessoa INT,
2                    nome VARCHAR,
3                    sobrenome VARCHAR,
4                    sexo CHAR,
5                    cpf CHAR) AS $$
6  BEGIN
7     RETURN QUERY SELECT * FROM pessoa_fisica;
8     RETURN;
9  END;
10 $$ LANGUAGE 'plpgsql'
11
12 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  CREATE TYPE type_pessoa_fisica AS (
2     id_pessoa INT,
3     nome VARCHAR,
4     sobrenome VARCHAR,
5     sexo CHAR,
6     cpf CHAR
7  );
```

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  CREATE FUNCTION blog_get_pessoas4() RETURNS SETOF type_pessoa_fisica AS $$
2  DECLARE
3     dados_pessoa type_pessoa_fisica;
4  BEGIN
5     FOR dados_pessoa IN SELECT id_pessoa, nome, sobrenome, sexo, cpf  FROM pessoa_fisica
6  LOOP
7        RETURN NEXT dados_pessoa;
8     END LOOP;
9     RETURN;
10 END;
11 $$ LANGUAGE 'plpgsql'
12
   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.

**Referências:**


- PostgreSQL Documentation – Create Function
- WikiBooks – Funções em PL/pgSQL
- PostgreSQL Brasil – Retornando registros de consultas genéricas com RETURN QUERY
- 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