

## PostgreSQL, как сохранить SELECT запрос

SELECT запросы очень удобно сохранять как функции и/или хранимые процедуры в PostgreSQL. Можно дать им любое имя и вызывать когда нужно.

### Сохраняем простой SELECT запрос как функцию

Для примера, представим, что у нас есть таблица `app_user`, которая хранит всех пользователей нашего приложения.

Если тебе нужно получить количество пользователей твоего приложения, ты напишешь что-то вроде:

```
SELECT count(*) FROM app_user;
```

Чтобы дать имя этому SQL запросу, ты можешь сохранить его как функцию.

```
create or replace function count_app_users()
  returns table
  (
    total bigint
  )
  language plpgsql
as
$$
begin
  return query (
    select count(*) from app_user
  );
end;
$$;
```

Давай разберем пример по шагам:

- В первой строке мы создаем функцию и называем ее `count_app_users`.
- Команда `create or replace` используется для того, чтобы создать функцию, если она не существует, или обновить ее, если она уже есть. Если ты уверен в том, что функции еще нет, можешь использовать обычный `create`
- Дальше мы явно указываем тип результата. Она равен `table` для всех SELECT запросов.
- В скобках мы указываем столбцы итоговой таблицы и их тип.
- Команду `language plpgsql` и несколько следующих строк ты можешь рассматривать как шаблон и не слишком вникать в них
- Сам запрос мы помещаем внутрь выражения `return query(...)` которое обернуто в `$$ begin` и `end $$`

### Вызываем функцию

После того, как мы сохранили SELECT запрос, мы можем вызвать его.

```
SELECT * FROM count_app_users();
```

Такой синтаксис получается потому, что процедура возвращает таблицу. Чтобы получить все ее строки мы выполняем обычный `SELECT *`

## Передаем аргументы в функцию PostgreSQL

Дальше, представим, что мы хотим посчитать только пользователей созданных после определенной даты.

Чтобы сделать это, нам нужно передать параметра внутрь функции и использовать его внутри SELECT запроса. В сигнатуру процедуры, мы добавим имя параметров и их типы. Потом, внутри SELECT, мы обратимся к ним.

```
create or replace function count_app_users(after timestamp) -- declaring the "after"
--parameter
returns table
(
    total bigint
)
language plpgsql
as
$$
begin
    return query (
        select count(*) from app_user
        where created_at > af -- using the "after" parameter
    );
end;
$$;
```

Теперь мы можем передать дату в процедуру count\_app\_users и получить количество всех пользователей созданных после этой даты.

```
SELECT * FROM count_app_users('2020-11-01');
```

## Синтаксис функции для вывода запроса как таблицы

```
CREATE OR REPLACE FUNCTION имя_функции(арг1 тип, арг2 тип,...)
    RETURNS TABLE(имя_поля1 тип, имя_поля2 тип, ...) AS $$
-- тип "имя_поляX" соответствует типу "atrX"
BEGIN
    RETURN QUERY
    SELECT atr1, atr2, ... FROM tabl1
    WHERE tabl1.atr2 > арг1 AND tabl1.atr1 = арг2; --пример запроса
END;
$$ LANGUAGE plpgsql;
```

### Пример:

```
-- Создана таблица
CREATE TABLE IF NOT EXISTS public.orders
(
    order_id integer NOT NULL,
    product_no integer,
    quantity integer,
    date_ord date,
    CONSTRAINT orders_pkey PRIMARY KEY (order_id),
```

CONSTRAINT orders\_product\_no\_fkey FOREIGN KEY (product\_no) )

Данные в таблице orders:

Data output   Сообщения   Notifications					
	order_id [PK] integer	product_no integer	quantity integer	date_ord date	
1	1	1	5	2022-09-01	
2	2	2	7	2022-09-02	
3	3	1	3	2022-09-03	
4	4	1	10	2022-09-03	
5	5	2	6	2022-09-04	

Функция `get_order_query_date(ord_data DATE)` выводит записи из таблицы `orders` позже некоторой даты, аргументу `ord_data` - дата заказа:

```
CREATE OR REPLACE FUNCTION get_order_query_date(ord_data DATE)
RETURNS TABLE(id_заказа int, №_товара int, Кол_во int, Дата_заказа DATE) AS $$
BEGIN
RETURN QUERY
select * from orders WHERE orders.date_ord > ord_data;
END;
$$ LANGUAGE plpgsql;
```

Вызов функции, чтобы получить результат запроса `SELECT`  
`SELECT * FROM get_order_query_date('2022-09-01');`

	id_заказа integer	№_товара integer	Кол_во integer	Дата_заказа date	
1	2	2	7	2022-09-02	
2	3	1	3	2022-09-03	
3	4	1	10	2022-09-03	
4	5	2	6	2022-09-04	

## Удаление функции PostgreSQL

Если ты попробуешь вызывать функцию `count_app_users` без аргументов, она будет до сих пор работать. Это получается потому, что функция определяется не только именем, но и типом и количеством параметров.

И у нас получилось две функции. Первая — `count_app_users`, вторая — `count_app_users(after)`. Обе они отлично живут в SQL базе данных.

Если одна из них тебе не нужна, то ее можно удалить так:

```
DROP function count_app_users();
```

Синтаксис похож на удаление таблицы и в нем нет каких-то подводных камней.

### **Посмотреть текст функции PostgreSQL**

Наш пример с count намеренно максимально упрощен, но если у тебя более длинная функция, то бывает полезно получить ее текст. Например, чтобы понять, как она работает или чтобы дальше его изменить.

Чтобы получить текст функции процедуры в PostgreSQL, тебе нужно знать ее имя:

```
SELECT prosrc
FROM pg_proc
WHERE proname = 'count_app_users';
```

Такой запрос вернет нам все, что мы писали между begin и end при создании.

### **Разбираемся с частыми ошибками**

Первая ошибка, с которой ты можешь столкнуться при сохранении запроса SELECT, выглядит примерно так:

```
ERROR: query has no destination for result data
Hint: If you want to discard the results of a SELECT, use PERFORM instead.
Where: PL/pgSQL function count_app_users() line 3 at SQL statement
```

Она возникает если ты не оберешь SELECT запрос в выражение

Дальше, может быть ошибка, связанная с неправильным вызовом функции или хранимой процедуры процедуры в PostgreSQL:

```
ERROR: cannot use RETURN QUERY in a non-SETOF function
```

Скорее всего ты попробовал вызывать функцию так:

```
call count_app_users();
```

Или так:

```
SELECT count_app_users();
```

Так как наша функция возвращала таблицу, то и результат из этой таблицы нужно получать как из любых других с помощью SELECT.

```
SELECT * FROM count_app_users();
```