




# Онлайн-образование



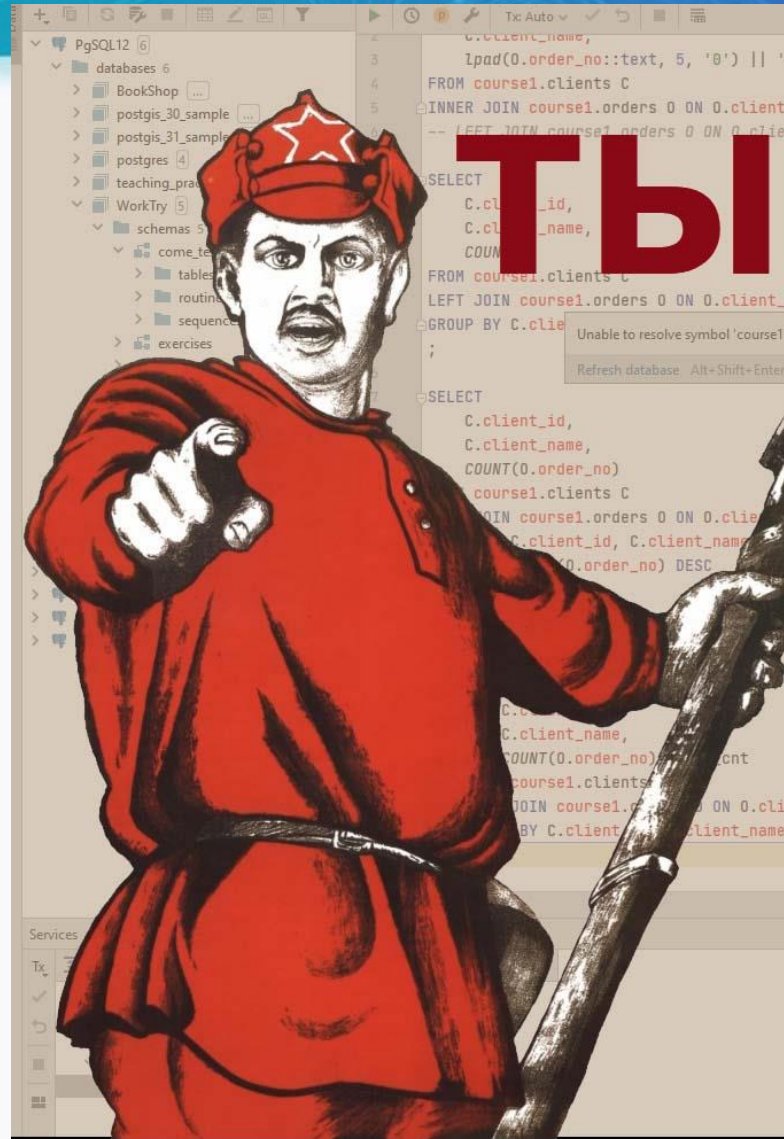


# Меня хорошо видно && слышно?

Ставьте  , если все хорошо  
Напишите в чат, если есть проблемы  
заодно проверяем, включена ли запись занятия



...а включена ли запись?



ВКЛЮЧИЛ ЗАПИСЬ?





# Хранимые функции и процедуры



ЗОЛОТОВ АНТОН

telegram @AVZolotov

# Правила вебинара



Активно участвуем



Задаем вопрос в чат или голосом



Off-topic обсуждаем вSlack



Вопросы вижу в чате, могу ответить не сразу



# Маршрут вебинара



# Цели вебинара | После занятия вы сможете

- 1 понять как работают хранимые функции и процедуры и для чего они нужны
- 2 объяснить назначение триггеров и курсоров
- 3 обрабатывать ошибки в хранимых функциях и процедурах

# Смысл | Зачем вам это уметь, в результате:

**1** настроить безопасность

**2** настроить оптимальную производительность

**3** научитесь использовать хранимые функции и процедуры





# Функции (SQL)



# Функции и процедуры в базе данных. Обзор

## Но зачем???

- Упрощение - возможность декомпозиции большой задачи, можно заниматься разработкой, абстрагируясь от общей задачи.
- Сокращение объёма передаваемых клиенту данных.
- Отделение интерфейса от реализации
  - *Разработчик клиентского приложения может мыслить в терминах прикладной задачи не отвлекаясь на изучения деталей хранения данных в БД.*
  - *Снижается вероятность появления в БД некорректных данных.*
  - *Меньше вероятность, что при внесении изменений в БД (рефакторинг, оптимизация) придется переписывать клиентские программы.*
- Безопасность: у пользователя есть права на выполнения функции/процедуры, но нет доступа к таблицам.



# Функции и процедуры в базе данных. Обзор

## Инструментарий

Встроенные языки		"Классические" языки программирования	Скриптовые языки	Язык статистических расчётов R
SQL	PL/PgSQL	C, C++, Java	PL/Perl, PL/Python, PL/Tcl - в стандартной поставке	PL/R - При установке модуля PL/R
			PL/Ruby, PL/Lua, PL/LOLCODE, PL/sh, PL/Scheme, PL/PHP, PL/v8 (Javascript) - при установке расширений	

<https://habr.com/ru/company/postgrespro/blog/502254/>

# Функции. Создание

## Что же это такое?

Объект базы данных – создается в конкретной схеме, определение хранится в системном каталоге (см. pg\_catalog.pg\_proc)

```
CREATE [ OR REPLACE ] FUNCTION имя
(
  [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ { DEFAULT | = }
выражение_по_умолчанию ] [, ...] ]
)
  [ RETURNS тип_результата
    | RETURNS TABLE ( имя_столбца тип_столбца [, ...] ) ]
{ LANGUAGE имя_языка
  ...
  AS 'определение'
} ...
```



# Функции. Создание

Определение, т.е. собственно код (тело) функции на одном из допустимых языков (интерпретируемых), представляет собой текстовую строку. Удобнее заключать её не в апострофы, а в «скобки-доллары» (см. [Подраздел 4.1.2.4](#)) - это позволяет при дальнейшем написании кода не задумываться об экранировании символа апострофа

```
CREATE OR REPLACE FUNCTION do_nothing()  
RETURNS void  
AS  
'SELECT NULL;'  
LANGUAGE sql;
```

```
CREATE OR REPLACE FUNCTION do_text()  
RETURNS text  
AS  
$BODY$  
    SELECT '1234567';  
$BODY$  
LANGUAGE sql;
```

*функция уже не "void" - она возвращает строку (text)*

# Функции. Аргументы

## Аргументы

```
function (  
    [режим] [имя] тип_данных,      -- параметр 1  
    [режим] [имя] тип_данных,      -- параметр 2  
    ...  
)  
return тип_результата
```

### режим

- IN (входной),
- OUT (выходной),
- INOUT (входной и выходной)
- VARIADIC (переменный).

По умолчанию подразумевается IN. За единственным аргументом VARIADIC могут следовать только аргументы OUT. Кроме того, аргументы OUT и INOUT нельзя использовать с предложением RETURNS TABLE.



# Функции. Аргументы

## Тип результата

- Может быть базовым или доменным типом, ссылкой на столбец таблицы (записывается как **имя\_таблицы.имя\_столбца%TYPE**)\*
- Выражение RETURNS может быть опущено в случае наличия OUT или INOUT аргументов. Если оно присутствует – должно согласовываться с типом результата, выводимым из выходных аргументов; если выходных аргументов несколько - в качестве типа результата указывается RECORD
- Если функция возвращает множество – перед типом данных указывается SETOF
- Функции, возвращающие отношение объявляются, как RETURNS TABLE (или RETURNS SETOF \*\*\*)
- Если функция ничего не возвращает – в качестве типа результата указывается void

*\* В зависимости от языка реализации здесь также могут допускаться «псевдотипы», например cstring.*

# Функции. Передача значений аргументов

## Позиционная передача

Традиционный механизм передачи аргументов – значения указываются в заданном порядке

```
SELECT some_function ('Hello, world!', 99, true);
```

## Именная передача

При именной передаче для аргумента добавляется имя, которое отделяется от выражения значения знаками =>

```
SELECT some_function (param_string => 'Hello, world!', param_bool => true);
```

## Смешанная передача

Параметры передаются и по именам, и по позиции. Именованные аргументы не могут стоять перед позиционными

```
SELECT some_function ('Hello, world!', param_bool => true);
```

*Именная и смешанная передача в настоящий момент не может использоваться при вызове агрегатной функции (но они допускаются, если агрегатная функция используется в качестве оконной).*



# Функции. Перегрузка

PostgreSQL позволяет определить несколько функций с одним и тем же именем, т.е. перегружать функции. При выполнении запроса, содержащего обращение к функции, сервер определяет, какую именно функцию следует вызвать, по количеству и типам представленных аргументов.

Создавая семейство перегруженных функций, необходимо не допускать неоднозначности.

Например, если созданы функции:

```
CREATE FUNCTION test(int, real) RETURNS ...
```

```
CREATE FUNCTION test(smallint, double precision) RETURNS ...
```

не вполне понятно, какая функция будет вызвана с довольно простыми аргументами вроде test(1, 1.5).

# Функции. Полиморфизм

Функции могут быть объявлены как принимающие и возвращающие полиморфные типы, описанные в [Подразделе 37.2.5.](#)

Пример:

функция, создающая массив из двух элементов произвольных (но одинаковых!) типов:

```
CREATE FUNCTION make_array(anyelement, anyelement)
RETURNS anyarray AS
$$
    SELECT ARRAY[$1, $2];
$$ LANGUAGE SQL;
```

```
SELECT make_array(1, 2) AS intarray, make_array('a'::text, 'b') AS textarray;
```

intarray	textarray
{1,2}	{a,b}

# Функции. Полиморфизм

Начиная с 13-й версии PG можно определить функцию, получающую на входе аргументы различных (но совместимых!) типов:

```
CREATE FUNCTION make_array_2(anycompatible, anycompatible)
RETURNS anycompatiblearray
AS
$$
    SELECT ARRAY[$1, $2];
$$ LANGUAGE SQL;
```

```
SELECT make_array_2(1, 2.5) AS intarray, make_array_2('a', 'b') AS textarray;
```

intarray	textarray
{1,2.5}	{a,b}



# Функции. Полиморфизм

Функция с полиморфными аргументами может иметь фиксированный тип результата, однако обратное не допускается. Например:

```
CREATE FUNCTION is_greater(anyelement, anyelement)
RETURNS boolean AS
$$
    SELECT $1 > $2;
$$ LANGUAGE SQL;
```

```
SELECT is_greater(1, 2);    -- OK
```

```
CREATE FUNCTION invalid_func()
RETURNS anyelement AS
$$
    SELECT 1;
$$ LANGUAGE SQL;
```

```
ERROR: cannot determine result data type
DETAIL: A result of type anyelement requires
```

# Функции. Категории изменчивости

Категория изменчивости – своего рода «обещание» некоторого поведения функции для оптимизатора.

Может принимать одно из значений:

<b>VOLATILE</b>	Возвращаемое значение может произвольно меняться при одинаковых значениях входных параметров, следовательно, она должна вычисляться при каждом вызове. Функция может менять состояние базы данных. <i>Используется по умолчанию.</i>
<b>STABLE</b>	Возвращаемое значение не меняется в пределах одного оператора SQL при неизменных значениях входных параметров. Функция не может менять состояние базы данных. Функция <b>может быть</b> выполнена однократно во время выполнения запроса, затем, вместо вызова функции будет подставляться вычисленное значение.
<b>IMMUTABLE</b>	Возвращаемое значение зависит только от входных параметров. Функция <b>может быть</b> выполнена на этапе планирования запроса, а не на этапе выполнения.

# Функции. Правила вызова с NULL-аргументами

Может принимать одно из значений:

<b>CALLED ON NULL INPUT</b>	функция будет вызвана как обычно, если среди её аргументов оказываются значения NULL. В этом случае ответственность за проверку значений NULL и соответствующую их обработку ложится на разработчика функции. <i>Используется по умолчанию.</i>
<b>RETURNS NULL ON NULL INPUT</b>	Функция всегда возвращает NULL, получив NULL в одном из аргументов. Такая функция не будет вызываться с аргументами NULL, вместо этого автоматически будет полагаться результат NULL
<b>STRICT</b>	



# Функции. Возможности параллельного выполнения

## PARALLEL

Может принимать одно из значений:

<b>UNSAFE</b>	Функцию нельзя выполнять в параллельном режиме и присутствие такой функции в операторе SQL приводит к выбору последовательного плана выполнения. <i>Используется по умолчанию.</i>
<b>RESTRICTED</b>	Функцию можно выполнять в параллельном режиме, но только в ведущем процессе группы
<b>SAFE</b>	Функция безопасна для выполнения в параллельном режиме без ограничений.

# Функции. Права доступа к объектам БД

## SECURITY

Может принимать одно из значений:

<b>SECURITY INVOKER</b>	Функция будет выполняться с правами пользователя, вызвавшего её. <i>Используется по умолчанию.</i>
<b>SECURITY DEFINER</b>	Функция выполняется с правами владельца.

*Использование SECURITY DEFINER требует внимательного отношения к процедуре переноса функции, например при развертывании. На PROD-кластере может не оказаться пользователя, который являлся владельцем функции на DEV.*

# Функции. Права доступа к объектам БД

Так как функция SECURITY DEFINER выполняется с правами пользователя, владеющего ей, необходимо позаботиться о том, чтобы её нельзя было использовать не по назначению. В целях безопасности в пути search\_path следует исключить любые схемы, доступные на запись недоверенным пользователям. Это не позволит злонамеренным пользователям создать свои объекты (например, таблицы, функции и операторы), которые замаскируют объекты, используемые функцией. Особенно важно в этом отношении исключить схему временных таблиц, которая по умолчанию просматривается первой, а право записи в неё по умолчанию имеют все. Соответствующую защиту можно организовать, поместив временную схему в конец списка поиска.





Вопросы?

# Функции. Итоги

1. Что дает возможность создавать программные модули на стороне сервера?
2. Какие способы передачи значений аргументов используются в PG?
3. Какие характеристики изменчивости можно задать для функции и как характеристика изменчивости влияет на выполнение?



The background of the slide is a high-angle, blue-tinted aerial photograph of a dense urban skyline, likely New York City. Overlaid on this image is a semi-transparent blue band across the middle, which contains a white network diagram of interconnected nodes and lines. The title text is centered within this band.

# Обзор языка PL/PgSQL



# PL/PgSQL это...

процедурный язык для СУБД PostgreSQL. Целью проектирования PL/pgSQL было создание загружаемого процедурного языка, который:

- используется для создания функций и триггеров,
- добавляет управляющие структуры к языку SQL,
- может выполнять сложные вычисления,
- наследует все пользовательские типы, функции и операторы,
- прост в использовании.

Функции PL/pgSQL могут использоваться везде, где допустимы встроенные функции. Например, можно создать функции со сложными вычислениями и условной логикой, а затем использовать их при определении операторов или в индексных выражениях.

В версии PostgreSQL 9.0 и выше, PL/pgSQL устанавливается по умолчанию

<https://postgrespro.ru/docs/postgresql/12/plpgsql>

# PL/PgSQL. Функции.

Так же, как и функции на «чистом» SQL, функции на PL/PgSQL определяются на сервере командами **CREATE FUNCTION**.

Такая команда обычно выглядит, например, так:

```
CREATE FUNCTION somefunc(integer, text)  
RETURNS integer  
AS 'тело функции'  
LANGUAGE plpgsql;
```

и, так же как в случае использования языка SQL, тело функции представляет собой просто текстовую строку.

# PL/PgSQL понятие блока.

PL/pgSQL это блочно-структурированный язык. Текст тела функции должен быть *блоком*. Структура блока:

```
[ <<метка>> ]  
[ DECLARE  
  объявления ]  
BEGIN  
  операторы  
END [ метка ];
```

Каждое объявление и каждый оператор в блоке должны завершаться символом ";"(точка с запятой). Блок, вложенный в другой блок, должен иметь точку с запятой после END, как показано выше. Однако финальный END, завершающий тело функции, не требует точки с запятой.



# PL/PgSQL понятие блока.

*Метка* требуется только тогда, когда нужно идентифицировать блок в операторе EXIT, или дополнить имена переменных, объявленных в этом блоке. Если метка указана после END, то она должна совпадать с меткой в начале блока.

Ключевые слова не чувствительны к регистру символов. Как и в обычных SQL-командах, идентификаторы неявно преобразуются к нижнему регистру, если они не взяты в двойные кавычки.

Комментарии в PL/pgSQL коде работают так же, как и в обычном SQL. Двойное тире (--) начинает комментарий, который завершается в конце строки. Блочный комментарий начинается с /\* и завершается \*/. Блочные комментарии могут быть вложенными.

# PL/PgSQL понятие блока.

Любой оператор в выполняемой секции блока может быть *вложенным блоком*. Вложенные блоки используются для логической группировки нескольких операторов или локализации области действия переменных для группы операторов. Во время выполнения вложенного блока переменные, объявленные в нём, скрывают переменные внешних блоков с такими же именами. Чтобы получить доступ к внешним переменным, нужно дополнить их имена меткой блока.

Например:

# PL/PgSQL понятие блока.

```
CREATE FUNCTION somefunc() RETURNS integer AS $$
<< outerblock >>
DECLARE
    quantity integer := 30;
BEGIN
    RAISE NOTICE 'Сейчас quantity = %', quantity; -- Выводится 30
    quantity := 50;
    --
    -- Вложенный блок
    --
    DECLARE
        quantity integer := 80;
    BEGIN
        RAISE NOTICE 'Сейчас quantity = %', quantity; -- Выводится 80
        RAISE NOTICE 'Во внешнем блоке quantity = %', outerblock.quantity; -- Выводится 50
    END;

    RAISE NOTICE 'Сейчас quantity = %', quantity; -- Выводится 50

    RETURN quantity;
END;
$$ LANGUAGE plpgsql;
```



# PL/PgSQL понятие блока.

Существует скрытый «внешний блок», окружающий тело каждой функции на PL/pgSQL.

Этот блок содержит объявления параметров функции (если они есть), а также некоторые специальные переменные, такие как FOUND (см. [Подраздел 42.5.5](#)). Этот блок имеет метку, совпадающую с именем функции, таким образом, параметры и специальные переменные могут быть дополнены именем функции.

# PL/PgSQL. Понятие блока.

Важно не путать использование BEGIN/END для группировки операторов в PL/pgSQL с одноимёнными SQL-командами для управления транзакциями. BEGIN/END в PL/pgSQL служат только для группировки предложений; они не начинают и не заканчивают транзакции. Управление транзакциями в PL/pgSQL описывается в [Разделе 42.8](#). Кроме того, блок с предложением EXCEPTION по сути создаёт вложенную транзакцию, которую можно отменить, не затрагивая внешнюю транзакцию. Подробнее это описано в [Подразделе 42.6.8](#).

Hint:

*у команды управления транзакцией **BEGIN** есть синоним – **START TRANSACTION***

# PL/PgSQL. Анонимный блок

Анонимный блок кода не принимает аргументы, а любое значение, которое он мог бы вернуть, отбрасывается. Он работает подобно коду функции которая не имеет параметров и возвращает void. Этот код разбирается и выполняется один раз.

Анонимный блок выполняется оператором DO

```
DO [ LANGUAGE имя_языка ] код
```

В отличие от функций для анонимных блоков определен язык по умолчанию: plpgsql

Пример:

```
DO
$$
DECLARE
    x          integer = 1;
BEGIN
    UPDATE some_table SET col_x = x
END
$$;
```



# PL/PgSQL. Переменные.

Все переменные, используемые в блоке, должны быть определены в секции объявления -DECLARE. (За исключением переменной-счётчика цикла FOR, которая объявляется автоматически. Для цикла по диапазону чисел автоматически объявляется целочисленная переменная, а для цикла по результатам курсора - переменная типа record.)

Переменные PL/pgSQL могут иметь любой тип данных SQL

Общий синтаксис объявления переменной:

***имя* [ CONSTANT ] *тип* [ COLLATE *имя\_правила\_сортировки* ] [ NOT NULL ] [ { DEFAULT | := | = } *выражение* ];**

<b>DEFAULT</b>	Задаёт начальное значение, которое присваивается переменной при входе в блок. Если отсутствует, то переменная инициализируется SQL-значением NULL.
<b>CONSTANT</b>	Предотвращает изменение значения переменной после инициализации, таким образом, значение остаётся постоянным в течение всего блока.
<b>COLLATE</b>	Определяет правило сортировки, которое будет использоваться для этой переменной
<b>NOT NULL</b>	Попытка присвоить NULL во время выполнения приведёт к ошибке. Все переменные, объявленные как NOT NULL, должны иметь непустые значения по умолчанию

# PL/PgSQL - Переменные

Примеры объявления переменных:

<b>user_id</b>	<b>CONSTANT integer = 10;</b>
<b>quantity</b>	<b>numeric(5) DEFAULT 144.1;</b>
<b>url</b>	<b>varchar := 'http://mysite.com';;</b>
<b>myrow</b>	<b>tablename%ROWTYPE;</b>
<b>myfield</b>	<b>tablename.columnname%TYPE;</b>

Значение по умолчанию вычисляется и присваивается переменной каждый раз при входе в блок (не только при первом вызове функции). Так, например, если переменная типа timestamp имеет функцию now() в качестве значения по умолчанию, это приведёт к тому, что переменная всегда будет содержать время текущего вызова функции, а не время, когда функция была предварительно скомпилирована.

# PL/PgSQL. Переменные составного типа – кортежи (записи)

Примеры объявления переменных составного типа:

```
имя      имя_составного_типа ;  
имя      имя_таблицы%ROWTYPE ;
```

Переменная составного типа называется строковой переменной (или переменной типа строки).

Значением такой переменной может быть целая строка, полученная в результате выполнения запроса SELECT или FOR, при условии, что набор столбцов запроса соответствует заявленному типу переменной. Доступ к отдельным значениям полей строковой переменной осуществляется, как обычно, через точку, например rowvar.field.

# PL/PgSQL. Переменные составного типа – кортежи (записи)

```
CREATE FUNCTION merge_fields(t_row table1)
RETURNS text
AS
$$
DECLARE
    t2_row table2%ROWTYPE;
BEGIN
    SELECT * INTO t2_row FROM table2 WHERE ... ;
    RETURN t_row.f1 || t2_row.f3 || t_row.f5 || t2_row.f7;
END;
$$ LANGUAGE plpgsql;

SELECT merge_fields(t.*) FROM table1 t WHERE ... ;
```



# Типы record

*имя* RECORD;

Переменные типа record похожи на переменные строкового ("ROW") типа, но они не имеют predetermined структуры. Они приобретают фактическую структуру от строки, которая им присваивается командами SELECT или FOR. Структура переменной типа record может меняться каждый раз при присвоении значения. Следствием этого является то, что пока значение не присвоено первый раз, переменная типа record не имеет структуры и любая попытка получить доступ к отдельному полю приведёт к ошибке во время исполнения.

# Типы record

Обратите внимание:

- RECORD это не подлинный тип данных, а только лишь заполнитель.
- Функция на PL/pgSQL, имеющая тип возвращаемого значения record, это не то же самое, что и переменная типа record, хотя такая функция может использовать переменную типа record для хранения своего результата. В обоих случаях фактическая структура строки неизвестна во время создания функции, но для функции, возвращающей record, фактическая структура определяется во время разбора вызывающего запроса, в то время как переменная типа record может менять свою структуру на лету.

# PL/PgSQL. Аргументы

Теперь стал окончательно понятен разобранный ранее пример задания параметров с использованием псевдонимов (ALIAS):

```
CREATE OR REPLACE FUNCTION add(integer, integer)
RETURNS integer
AS
$BODY$
DECLARE
    first_val    ALIAS FOR $1;
    second_val   ALIAS FOR $2;
BEGIN
    RETURN first_val + second_val;
END
$BODY$
LANGUAGE plpgsql;
```

# PL/PgSQL. Аргументы

Задание именованных аргументов и использование ALIAS в секции DECLARE не являются полностью эквивалентными.

Для именованного аргумента допустимо обращение **function\_name.variable\_name**, при использовании ALIAS такая ссылка невозможна. Но если для обрамляющего блока определена метка label, до можно использовать обращение **label.variable\_name**



# PL/PgSQL. Возвращаемые значения и выходные параметры

Выходные параметры наиболее полезны для возвращения нескольких значений. Простейший пример:

```
CREATE FUNCTION sum_n_product(x int, y int, OUT sum int, OUT prod int)
AS
$$
BEGIN
    sum = x + y;          -- out-параметрам присваиваются значения
    prod = x * y;
END;
$$ LANGUAGE plpgsql;
```

Здесь фактически создаётся анонимный тип record для возвращения результата функции. Если используется предложение RETURNS, то оно должна выглядеть как RETURNS record

# PL/PgSQL. Возвращаемые значения и выходные параметры

Как вернуть отношение?

Объявить функцию на PL/pgSQL с использованием RETURNS TABLE, например:

```
CREATE FUNCTION extended_sales(p_itemno int)
RETURNS TABLE(quantity int, total numeric)
AS
$$
BEGIN
    RETURN QUERY
    SELECT s.quantity, s.quantity * s.price FROM sales s WHERE s.itemno =
p_itemno;
END;
$$ LANGUAGE plpgsql;    -- если бы использовался SQL – "RETURN QUERY" не потребовался бы
```

Это в точности соответствует объявлению одного или нескольких параметров OUT и указанию RETURNS SETOF *некий\_тип*.

*Гарантировать порядок в SETOF мы не можем*





Вопросы?



The background of the slide is an aerial photograph of a dense city skyline, likely New York City, with numerous skyscrapers. The image is overlaid with a semi-transparent blue layer that features a white geometric network pattern of interconnected lines and dots. The title text is centered within this blue band.

# Выборка в функциях



# Основные операторы

## Выполнение команды, не возвращающей результат

В функции на PL/pgSQL можно выполнить любую команду SQL, не возвращающую строк, просто написав эту команду (например, INSERT без предложения RETURNING).

Иногда бывает полезно вычислить значение выражения или запроса SELECT, но отказаться от результата, например, при вызове функции, у которой есть побочные эффекты, но нет полезного результата. Для этого в PL/pgSQL, используется оператор PERFORM:

PERFORM *запрос*;

# Основные операторы

## Выполнение запроса, возвращающего одну строку

Результат SQL-команды, возвращающей одну строку (возможно из нескольких столбцов), может быть присвоен переменной типа `record`, переменной строкового типа или списку скалярных переменных. Для этого нужно к основной команде SQL добавить предложение `INTO`. Так, например:

```
SELECT выражения_select INTO [STRICT] цель FROM ...;  
INSERT ... RETURNING выражения INTO [STRICT] цель;  
UPDATE ... RETURNING выражения INTO [STRICT] цель;  
DELETE ... RETURNING выражения INTO [STRICT] цель;
```

Предложение `INTO` может появиться практически в любом месте SQL-команды. Обычно его записывают непосредственно перед или сразу после списка *выражения\_select* в `SELECT` или в конце команды для команд других типов. Рекомендуется следовать этому соглашению на случай, если правила разбора PL/pgSQL ужесточатся в будущих версиях.

# Основные операторы

## Выполнение запроса, возвращающего одну строку

Если указание **STRICT** **отсутствует** в предложении INTO, то **цели** **присваивается первая** строка, возвращённая запросом; или NULL, если запрос не вернул строк. (Заметим, что понятие «первая строка» определяется неоднозначно без ORDER BY.) Все остальные строки результата после первой отбрасываются. Можно проверить специальную переменную FOUND (см. [Подраздел 42.5.5](#)), чтобы определить, была ли возвращена запись:

```
SELECT * INTO myrec FROM emp WHERE empname = myname;  
IF NOT FOUND THEN  
    RAISE EXCEPTION 'Сотрудник % не найден', myname;  
END IF;
```

# Основные операторы

## Выполнение запроса, возвращающего одну строку

Если добавлено указание **STRICT**, то запрос должен вернуть **ровно одну строку** или произойдёт ошибка во время выполнения: либо NO\_DATA\_FOUND (нет строк), либо TOO\_MANY\_ROWS (более одной строки). Можно использовать секцию исключений в блоке для обработки ошибок, например:

```
BEGIN
  SELECT * INTO STRICT myrec FROM emp WHERE empname = myname;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RAISE EXCEPTION 'Сотрудник % не найден', myname;
  WHEN TOO_MANY_ROWS THEN
    RAISE EXCEPTION 'Сотрудник % уже существует', myname;
END;
```



# Основные операторы

## Не делать ничего

Иногда бывает полезен оператор, который не делает ничего. Например, он может показывать, что одна из ветвей if/then/else сознательно оставлена пустой. Для этих целей используется NULL:

```
NULL;
```

В следующем примере два фрагмента кода эквивалентны:

```
BEGIN
  y := x / 0;
EXCEPTION
  WHEN division_by_zero THEN
    NULL; -- ошибка игнорируется
END;

BEGIN
  y := x / 0;
EXCEPTION
  WHEN division_by_zero THEN -- ошибка игнорируется
END;
```

# Динамический SQL

## Выполнение динамически формируемых команд

Часто требуется динамически формировать команды внутри функций на PL/pgSQL, то есть такие команды, в которых при каждом выполнении могут использоваться разные таблицы или типы данных. Обычно PL/pgSQL кеширует планы выполнения (как описано в [Подразделе 42.11.2](#)), но в случае с динамическими командами это не будет работать. Для исполнения динамических команд предусмотрен оператор EXECUTE:

EXECUTE *строка-команды* [ INTO [STRICT] *цель* ] [ USING *выражение* [, ... ] ];

где *строка-команды* это выражение, формирующее строку (типа text) с текстом команды, которую нужно выполнить. Необязательная *цель* — это переменная-запись, переменная-кортеж или разделённый запятыми список простых переменных и полей записи/кортежа, куда будут помещены результаты команды. Необязательные выражения в USING формируют значения, которые будут вставлены в команду.

# Динамический SQL

## Выполнение динамически формируемых команд

В сформированном тексте команды замена имён переменных PL/pgSQL на их значения проводиться не будет. Все необходимые значения переменных должны быть вставлены в командную строку при её построении, либо нужно использовать параметры, как описано ниже.

Также, **нет никакого плана кеширования** для команд, выполняемых с помощью EXECUTE. Вместо этого план создаётся каждый раз при выполнении. Таким образом, строка команды может динамически создаваться внутри функции для выполнения действий с различными таблицами и столбцами.

# Динамический SQL

## Выполнение динамически формируемых команд

В тексте команды можно использовать значения параметров, ссылки на параметры обозначаются как \$1, \$2 и т. д. Эти символы указывают на значения, находящиеся в предложении USING. Такой метод зачастую предпочтительнее, чем вставка значений в команду в виде текста: он позволяет исключить во время исполнения дополнительные расходы на преобразования значений в текст и обратно, и **не открывает возможности для SQL-инъекций**, не требуя применять экранирование или кавычки для спецсимволов. Пример:

```
EXECUTE 'SELECT count(*) FROM mytable WHERE inserted_by = $1 AND inserted <= $2'  
  
INTO c  
  
USING checked_user, checked_date;
```



# Динамический SQL

## Выполнение динамически формируемых команд

Обратите внимание, что символы параметров можно использовать только вместо значений данных. **Если же требуется динамически формировать имена таблиц или столбцов, их необходимо вставлять в виде текста.** Например, если в предыдущем запросе необходимо динамически задавать имя таблицы, можно сделать следующее:

```
EXECUTE 'SELECT count(*) FROM '  
    || quote_ident(tabname)  
    || ' WHERE inserted_by = $1 AND inserted <= $2'  
INTO c  
USING checked_user, checked_date;
```

# Динамический SQL

## Выполнение динамически формируемых команд

В качестве более аккуратного решения, вместо имени таблиц или столбцов можно использовать указание формата %I с функцией format() (текст, разделённый символами новой строки, соединяется вместе):

```
EXECUTE format('SELECT count(*) FROM %I '  
              'WHERE inserted_by = $1 AND inserted <= $2', tablename)  
INTO c  
USING checked_user, checked_date;
```

# Динамический SQL

## Использование кавычек в динамических запросах

При работе с динамическими командами часто приходится иметь дело с экранированием одинарных кавычек.

Можно напрямую вызывать функции заключения в кавычки:

```
EXECUTE 'UPDATE tbl SET '
```

```
|| quote_ident(colname)
```

```
|| ' = '
```

```
|| quote_literal(newvalue)
```

```
|| ' WHERE key = '
```

```
|| quote_literal(keyvalue);
```

# Динамический SQL

## Использование кавычек в динамических запросах

Динамические операторы SQL также можно безопасно сформировать, используя функцию `format` (см. [Подраздел 9.4.1](#)). Например:

```
EXECUTE format('UPDATE tbl SET %I = %L '  
              'WHERE key = %L', colname, newvalue, keyvalue);
```





Вопросы?

# Транзакции

<https://postgrespro.ru/docs/postgresql/12/plpgsql-transactions>



# Практика

<https://postgrespro.ru/docs/postgrespro/12/demodb-bookings>

<https://postgrespro.ru/education/demodb>

The background of the entire image is an aerial photograph of a city, likely New York City, showing a dense cluster of skyscrapers. A semi-transparent blue overlay covers the entire image. In the center, there is a horizontal band with a gradient from teal on the left to dark blue on the right. Overlaid on this band is a white network pattern of dots connected by thin lines. The title text is centered within this band.

# Управляющие конструкции



# Управляющие структуры

- Наиболее полезная и важная часть PL/pgSQL.
- С их помощью можно очень гибко и эффективно манипулировать данными PostgreSQL

<https://postgrespro.ru/docs/postgresql/12/plpgsql-control-structures>

## Команды для возврата значения из функции RETURN и RETURN NEXT

**RETURN** выражение;

RETURN с последующим выражением прекращает выполнение функции и возвращает значение выражения в вызывающую программу. Эта форма используется для функций PL/pgSQL, которые не возвращают набор строк.

В функции, возвращающей **скалярный тип**, результирующее выражение автоматически приводится к типу возвращаемого значения. Однако, чтобы вернуть составной тип (строку), возвращаемое выражение должно в точности содержать требуемый набор столбцов. При этом может потребоваться явное приведение типов.

Для функции с **выходными параметрами просто** используйте **RETURN без выражения**. Будут возвращены текущие значения выходных параметров.

Для функции, возвращающей void, RETURN можно использовать в любом месте, но без выражения после RETURN.

## Команды для возврата значения из функции RETURN и RETURN NEXT

Возвращаемое значение функции не может остаться не определённым. **Если достигнут конец блока верхнего уровня, а оператор RETURN так и не встретился**, происходит **ошибка** времени исполнения. Это не касается функций с выходными параметрами и функций, возвращающих void. Для них оператор RETURN выполняется автоматически по окончании блока верхнего уровня.

Несколько примеров:

-- Функции, возвращающие скалярный тип данных

```
RETURN 1 + 2;
```

```
RETURN scalar_var;
```

-- Функции, возвращающие составной тип данных

```
RETURN composite_type_var;
```

```
RETURN (1, 2, 'three'::text); -- требуется приведение типов
```

## RETURN NEXT и RETURN QUERY

**RETURN NEXT** *выражение*;

**RETURN QUERY** *запрос*;

**RETURN QUERY EXECUTE** *строка-команды* [USING *выражение* [, ...]];

Для **функций** на PL/pgSQL, возвращающих **SETOF** *некий\_тип*, нужно действовать несколько по-иному. Отдельные элементы возвращаемого значения формируются командами RETURN NEXT или RETURN QUERY, а финальная команда RETURN без аргументов завершает выполнение функции. RETURN NEXT используется как со скалярными, так и с составными типами данных. Для составного типа результат функции возвращается в виде таблицы. RETURN QUERY добавляет результат выполнения запроса к результату функции. RETURN **NEXT** и RETURN **QUERY** можно свободно **смешивать** в теле функции, в этом случае их **результаты** будут **объединены**.



# RETURN NEXT

RETURN NEXT и RETURN QUERY **не выполняют возврат** из функции. **Они** просто **добавляют строки в результирующее множество**. Затем выполнение продолжается со следующего оператора в функции. Успешное выполнение RETURN NEXT и RETURN QUERY формирует множество строк результата. Для **выхода** из функции используется **RETURN**, обязательно **без аргументов** (или **можно** просто **дождаться** окончания **выполнения** функции).

RETURN QUERY имеет разновидность RETURN QUERY EXECUTE, предназначенную для динамического выполнения запроса. В текст запроса можно добавить параметры, используя USING, так же как и с командой EXECUTE.

# RETURN NEXT

Для функции с выходными параметрами просто используйте RETURN NEXT без аргументов. При каждом исполнении RETURN NEXT текущие значения выходных параметров сохраняются для последующего возврата в качестве строки результата. Обратите внимание, что если функция с выходными параметрами должна возвращать множество значений, то при объявлении нужно указывать RETURNS SETOF. При этом если выходных параметров несколько, то используется RETURNS SETOF record, а если только один с типом *некий\_min*, то RETURNS SETOF *некий\_min*.

## Примечание

В текущей реализации RETURN NEXT и RETURN QUERY результирующее множество накапливается целиком, прежде чем будет возвращено из функции. Если множество очень большое, то это может отрицательно сказаться на производительности, так как при нехватке оперативной памяти данные записываются на диск. В следующих версиях PL/pgSQL это ограничение будет снято. В настоящее время управлять количеством оперативной памяти в подобных случаях можно параметром конфигурации [work\\_mem](#). При наличии свободной памяти администраторы должны рассмотреть возможность увеличения значения данного параметра.

# Условные операторы

Операторы IF и CASE позволяют выполнять команды в зависимости от определённых условий. PL/pgSQL поддерживает три формы IF:

- IF ... THEN ... END IF
- IF ... THEN ... ELSE ... END IF
- IF ... THEN ... ELSIF ... THEN ... ELSE ... END IF

и две формы CASE:

- CASE ... WHEN ... THEN ... ELSE ... END CASE
- CASE WHEN ... THEN ... ELSE ... END CASE



# IF-THEN

```
IF логическое-выражение THEN  
    операторы  
END IF;
```

IF-THEN это простейшая форма IF. Операторы между THEN и END IF выполняются, если условие (*логическое-выражение*) истинно. В противном случае они опускаются.

Пример:

```
IF v_user_id <> 0 THEN  
    UPDATE users SET email = v_email WHERE user_id = v_user_id;  
END IF;
```

## IF-THEN-ELSE

```
IF логическое-выражение THEN  
    операторы  
ELSE  
    операторы  
END IF;
```

IF-THEN-ELSE добавляет к IF-THEN возможность указать альтернативный набор операторов, которые будут выполнены, если условие не истинно (в том числе, если условие NULL).

# IF-THEN-ELSE

Примеры:

```
IF parentid IS NULL OR parentid = ''  
THEN  
    RETURN fullname;  
ELSE  
    RETURN hp_true_filename(parentid) || '/' || fullname;  
END IF;  
  
IF v_count > 0 THEN  
    INSERT INTO users_count(count) VALUES (v_count);  
    RETURN 't';  
ELSE  
    RETURN 'f';  
END IF;
```

# IF-THEN-ELSIF

IF *логическое-выражение* THEN  
    *операторы*

[ELSIF *логическое-выражение* THEN *операторы* [ELSIF *логическое-выражение* THEN  
    *операторы* ...]]

[ELSE *операторы*]

END IF;

В некоторых случаях двух альтернатив недостаточно. IF-THEN-ELSIF обеспечивает удобный способ проверки нескольких вариантов по очереди. Условия в IF последовательно проверяются до тех пор, пока не будет найдено первое истинное. После этого операторы, относящиеся к этому условию, выполняются, и управление переходит к следующей после END IF команде. (Все последующие условия не проверяются.) Если ни одно из условий IF не является истинным, то выполняется блок ELSE (если присутствует).



# IF-THEN-ELSIF

Пример:

```
IF number = 0 THEN  
    result := 'zero';  
ELSIF number > 0 THEN  
    result := 'positive';  
ELSIF number < 0 THEN  
    result := 'negative';  
ELSE  
    -- остаётся только один вариант: number имеет значение NULL  
    result := 'NULL';  
END IF;
```

# IF-THEN-ELSIF

Вместо ключевого слова ELSIF можно использовать ELSEIF.

Другой вариант сделать то же самое, это использование вложенных операторов IF-THEN-ELSE, как в следующем примере:

```
IF demo_row.sex = 'm' THEN
    pretty_sex := 'man';
ELSE
    IF demo_row.sex = 'f' THEN
        pretty_sex := 'woman';
    END IF;
END IF;
```

*Однако это требует написания соответствующих END IF для каждого IF, что при наличии нескольких альтернатив делает код более громоздким, чем использование ELSIF.*

# Case

CASE *выражение-поиска*

WHEN *выражение* [, *выражение* [...]] THEN

*операторы*

[WHEN *выражение* [, *выражение* [...]] THEN *операторы* ...]

[ELSE *операторы*]

END CASE;

Простая форма CASE реализует условное выполнение на основе сравнения операндов. *Выражение-поиска* вычисляется (один раз) и последовательно сравнивается с каждым *выражением* в условиях WHEN. Если совпадение найдено, то выполняются соответствующие *операторы* и управление переходит к следующей после END CASE команде. (Все последующие выражения WHEN не проверяются.) Если совпадение не было найдено, то выполняются *операторы* в ELSE. Но если ELSE нет, то вызывается исключение CASE\_NOT\_FOUND.

# Case

Пример:

```
CASE x  
  WHEN 1, 2 THEN  
    msg := 'один или два';  
  ELSE  
    msg := 'значение, отличное от один или два';  
END CASE;
```



# Case с перебором условий

CASE

WHEN *логическое-выражение* THEN

*операторы*

[WHEN *логическое-выражение* THEN *операторы* ...]

[ELSE *операторы*]

END CASE;

Эта форма CASE реализует условное выполнение, основываясь на истинности логических условий. Каждое *логическое-выражение* в предложении WHEN вычисляется по порядку до тех пор, пока не будет найдено истинное. Затем выполняются соответствующие *операторы* и управление переходит к следующей после END CASE команде. (Все последующие выражения WHEN не проверяются.) Если ни одно из условий не окажется истинным, то выполняются *операторы* в ELSE. Но если ELSE нет, то вызывается исключение CASE\_NOT\_FOUND.

# Case с перебором условий

Пример:

```
CASE  
  WHEN x BETWEEN 0 AND 10 THEN  
    msg := 'значение в диапазоне между 0 и 10';  
  WHEN x BETWEEN 11 AND 20 THEN  
    msg := 'значение в диапазоне между 11 и 20';  
END CASE;
```

Эта форма CASE полностью эквивалента IF-THEN-ELSIF, за исключением того, что при невыполнении всех условий и отсутствии ELSE, IF-THEN-ELSIF ничего не делает, а CASE вызывает ошибку.

# Простые циклы

Операторы

- LOOP,
- EXIT,
- CONTINUE,
- WHILE,
- FOR,
- FOREACH

позволяют повторить серию команд в функции на PL/pgSQL.

# Loop

```
[<<метка>>]
```

```
LOOP
```

```
    операторы
```

```
END LOOP [ метка ];
```

LOOP организует безусловный цикл, который повторяется до бесконечности, пока не будет прекращён операторами EXIT или RETURN. Для вложенных циклов можно использовать *метку* в операторах EXIT и CONTINUE, чтобы указать, к какому циклу эти операторы относятся.



# EXIT

EXIT [ *метка* ] [WHEN *логическое-выражение*];

Если *метка* не указана, то завершается самый внутренний цикл, далее выполняется оператор, следующий за END LOOP. Если *метка* указана, то она должна относиться к текущему или внешнему циклу, или это может быть метка блока. При этом в именованном цикле/блоке выполнение прекращается, а управление переходит к следующему оператору после соответствующего END.

При наличии WHEN цикл прекращается, только если *логическое-выражение* истинно. В противном случае управление переходит к оператору, следующему за EXIT.

EXIT можно использовать со всеми типами циклов, не только с безусловным.

Когда EXIT используется для выхода из блока, управление переходит к следующему оператору после окончания блока. **Обратите внимание, что для выхода из блока нужно обязательно указывать метку. EXIT без метки не позволяет прекратить работу блока.**

# EXIT

Примеры:

LOOP

```
-- здесь производятся вычисления  
IF count > 0 THEN  
    EXIT; -- выход из цикла  
END IF;  
END LOOP;
```

LOOP

```
-- здесь производятся вычисления  
EXIT WHEN count > 0; -- аналогично предыдущему примеру  
END LOOP;
```

<<ablock>>

BEGIN

```
-- здесь производятся вычисления  
IF stocks > 100000 THEN  
    EXIT ablock; -- выход из блока BEGIN  
END IF;  
-- вычисления не будут выполнены, если stocks > 100000  
END;
```

# CONTINUE

CONTINUE [ *метка* ] [WHEN *логическое-выражение*];

Если *метка* не указана, то начинается следующая итерация самого внутреннего цикла. То есть все оставшиеся в цикле операторы пропускаются, и управление переходит к управляющему выражению цикла (если есть) для определения, нужна ли ещё одна итерация цикла. Если *метка* присутствует, то она указывает на метку цикла, выполнение которого будет продолжено.

При наличии WHEN следующая итерация цикла начинается только тогда, когда *логическое-выражение* истинно. В противном случае управление переходит к оператору, следующему за CONTINUE.

CONTINUE можно использовать со всеми типами циклов, не только с безусловным.

# CONTINUE

Примеры:

LOOP

-- здесь производятся вычисления

EXIT WHEN count > 100;

CONTINUE WHEN count < 50;

-- вычисления для count в диапазоне 50 .. 100

END LOOP;



# WHILE

[<<метка>>]

WHILE *логическое-выражение* LOOP

*операторы*

END LOOP [ *метка* ];

WHILE выполняет серию команд до тех пор, пока истинно *логическое-выражение*.  
Выражение проверяется непосредственно перед каждым входом в тело цикла.

Пример:

WHILE amount\_owed > 0 AND gift\_certificate\_balance > 0 LOOP

-- здесь производятся вычисления

END LOOP;

WHILE NOT done LOOP

-- здесь производятся вычисления

END LOOP;

## FOR (целочисленный вариант)

[<<метка>>]

FOR *имя* IN [REVERSE] *выражение* .. *выражение* [BY *выражение*] LOOP

*операторы*

END LOOP [*метка*];

В этой форме цикла FOR итерации выполняются по диапазону целых чисел. Переменная *имя* автоматически определяется с типом integer и существует только внутри цикла (если уже существует переменная с таким именем, то внутри цикла она будет игнорироваться). Выражения для нижней и верхней границы диапазона чисел вычисляются один раз при входе в цикл. Если не указано BY, то шаг итерации 1, в противном случае используется значение в BY, которое вычисляется, опять же, один раз при входе в цикл. Если указано REVERSE, то после каждой итерации величина шага вычитается, а не добавляется.

# FOR

Примеры целочисленного FOR:

```
FOR i IN 1..10 LOOP
```

-- внутри цикла переменная i будет иметь значения 1,2,3,4,5,6,7,8,9,10  
END LOOP;

```
FOR i IN REVERSE 10..1 LOOP
```

-- внутри цикла переменная i будет иметь значения 10,9,8,7,6,5,4,3,2,1  
END LOOP;

```
FOR i IN REVERSE 10..1 BY 2 LOOP
```

-- внутри цикла переменная i будет иметь значения 10,8,6,4,2  
END LOOP;

Если нижняя граница цикла больше верхней границы (или меньше, в случае REVERSE), то тело цикла не выполняется вообще. При этом ошибка не возникает.

# Цикл по результатам запроса

Другой вариант FOR позволяет организовать цикл по результатам запроса. Синтаксис:

```
[ <<метка>> ]  
FOR цель IN запрос LOOP  
    операторы  
END LOOP [ метка ];
```

Переменная *цель* может быть строковой переменной, переменной типа record или разделённым запятыми списком скалярных переменных. Переменной *цель* последовательно присваиваются строки результата запроса, и для каждой строки выполняется тело цикла.

рассмотрим пример



# Цикл по результатам запроса

Для переменных PL/pgSQL в тексте запроса выполняется подстановка значений, план запроса кешируется для возможного повторного использования, как подробно описано в [Подразделе 42.11.1](#) и [Подразделе 42.11.2](#).

Ещё одна разновидность этого типа цикла FOR-IN-EXECUTE:

```
[ <<метка>> ]  
FOR цель IN EXECUTE выражение_проверки [ USING выражение [, ...] ] LOOP  
    операторы  
END LOOP [ метка ];
```

Она похожа на предыдущую форму, за исключением того, что текст запроса указывается в виде строкового выражения. Текст запроса формируется и для него строится план выполнения при каждом входе в цикл. Это даёт программисту выбор между скоростью предварительно разобранного запроса и гибкостью динамического запроса, так же, как и в случае с обычным оператором EXECUTE. Как и в EXECUTE, значения параметров могут быть добавлены в команду с использованием USING.

Ещё один способ организовать цикл по результатам запроса это объявить курсор. Описание в [Подразделе 42.7.4](#).

## Цикл по элементам массива

Цикл FOREACH очень похож на FOR. Отличие в том, что вместо перебора строк SQL-запроса происходит перебор элементов массива. Синтаксис цикла FOREACH:

```
[ <<метка>> ]  
FOREACH цель [ SLICE число ] IN ARRAY выражение LOOP  
    операторы  
END LOOP [ метка ];
```

Без указания SLICE, или если SLICE равен 0, цикл выполняется по всем элементам массива, полученного из *выражения*. Переменной *цель* последовательно присваивается каждый элемент массива и для него выполняется тело цикла.

# PL/PgSQL. Аргументы VARIADIC

Вернёмся к режимам аргументов – остался неразобранным режим VARIADIC

Функции SQL могут быть объявлены как принимающие переменное число аргументов, с условием, что все «необязательные» аргументы имеют один тип данных.

Необязательные аргументы будут переданы такой функции в виде массива. Для этого в объявлении функции последний параметр помечается как VARIADIC; при этом он должен иметь тип массива.



The image features a high-angle, aerial view of a dense urban skyline, likely New York City, with numerous skyscrapers and buildings. The entire image is overlaid with a semi-transparent blue and green gradient. A network of thin, light blue lines connects various points across the gradient, creating a digital or technological feel. In the center, the word "Вопросы?" is written in a large, white, sans-serif font.

**Вопросы?**





# Процедуры

# Процедуры

## Появились в PostgreSQL 11

### **CREATE [ OR REPLACE ] PROCEDURE**

```
имя ( [ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ {  
DEFAULT | = } выражение_по_умолчанию ] [, ...] )  
{ LANGUAGE имя_языка  
...  
} ...
```

Отличие состоит в том, что процедура не возвращает значение, и поэтому для неё не определяется возвращаемый тип.

Тогда как функция вызывается в составе запроса или команды DML, процедура вызывается явно, оператором [CALL](#)

<https://postgrespro.ru/docs/postgresql/12/sql-createprocedure>



## Завершение процедуры

Процедура не возвращает никакого значения, поэтому она может завершаться без оператора RETURN. Если вы хотите досрочно завершить выполнение кода оператором RETURN, напишите просто RETURN без возвращаемого выражения.

Если у процедуры есть выходные параметры, конечные значения соответствующих им переменных будут выданы вызывающему коду

## Вызов процедуры

Функция, процедура или блок DO в PL/pgSQL может вызвать процедуру, используя оператор CALL. Выходные параметры при этом обрабатываются не так, как это делает CALL в обычном SQL. Каждому параметру INOUT для процедуры должна соответствовать переменная в операторе CALL, и этой переменной по завершении процедуры будет присвоено возвращаемое процедурой значение.





Вопросы?





# Обработка ошибок

# Обработка ошибок

По умолчанию любая возникающая ошибка прерывает выполнение функции на PL/pgSQL, а также транзакцию, относящуюся к этой функции. Использование в блоке секции EXCEPTION позволяет перехватывать и обрабатывать ошибки. Синтаксис секции EXCEPTION расширяет синтаксис обычного блока:

```
[ <<метка>> ]  
[ DECLARE  
  объявления ]  
BEGIN  
  операторы  
EXCEPTION  
  WHEN условие [ OR условие ... ] THEN  
    операторы_обработчика  
  [ WHEN условие [ OR условие ... ] THEN  
    операторы_обработчика  
    ... ]  
END;
```



## Обработка ошибок

Если ошибок не было, то выполняются все *операторы* блока и управление переходит к следующему оператору после END. Но если при выполнении *оператора* происходит ошибка, то дальнейшая обработка прекращается и управление переходит к списку исключений в секции EXCEPTION. В этом списке ищется первое исключение, условие которого соответствует ошибке. Если исключение найдено, то выполняются соответствующие *операторы\_обработчика* и управление переходит к следующему оператору после END. Если исключение не найдено, то ошибка передаётся наружу, как будто секции EXCEPTION не было. При этом ошибку можно перехватить в секции EXCEPTION внешнего блока. Если ошибка так и не была перехвачена, то обработка функции прекращается.

В качестве *условия* может задаваться одно из имён, перечисленных в [Приложении А](#). Если задаётся имя категории, ему соответствуют все ошибки в данной категории. Специальному имени условия OTHERS (другие) соответствуют все типы ошибок, кроме QUERY\_CANCELED и ASSERT\_FAILURE. (И эти два типа ошибок можно перехватить по имени, но часто это неразумно.) Имена условий воспринимаются без учёта регистра



## Обработка ошибок

Условие ошибки также можно задать кодом SQLSTATE; например, эти два варианта равнозначны:

```
WHEN division_by_zero THEN ...
```

```
WHEN SQLSTATE '22012' THEN ...
```

Если при выполнении *операторов\_обработчика* возникнет новая ошибка, то она не может быть перехвачена в этой секции EXCEPTION. Ошибка передаётся наружу и её можно перехватить в секции EXCEPTION внешнего блока.

## Обработка ошибок

При выполнении команд в секции EXCEPTION локальные переменные функции на PL/pgSQL сохраняют те значения, которые были на момент возникновения ошибки. Однако, будут отменены все изменения в базе данных, выполненные в блоке

*Наличие секции EXCEPTION значительно увеличивает накладные расходы на вход/выход из блока, поэтому не используйте EXCEPTION без надобности.*

рассмотрим пример

# Обработка исключений для команд UPDATE/INSERT

```
DO
$$
DECLARE
    n integer;
BEGIN
    INSERT INTO t(id) VALUES (3);
    SELECT id INTO STRICT n FROM t;
    RAISE NOTICE 'Оператор SELECT INTO выполнен';
EXCEPTION
    WHEN no_data_found THEN
        RAISE NOTICE 'Нет данных';
    WHEN too_many_rows THEN
        RAISE NOTICE 'Слишком много данных';
        RAISE NOTICE 'Строк в таблице: %', (SELECT count(*) FROM t);
END;
$;
```

## Получение информации об ошибке

При обработке исключений часто бывает необходимым получить детальную информацию о произошедшей ошибке. Для этого в PL/pgSQL есть два способа: использование специальных переменных и команда GET STACKED DIAGNOSTICS.

Внутри секции EXCEPTION специальная переменная SQLSTATE содержит код ошибки, для которой было вызвано исключение (список возможных кодов ошибок приведён в [Таблице A.1](#)). Специальная переменная SQLERRM содержит сообщение об ошибке, связанное с исключением. Эти переменные являются неопределёнными вне секции EXCEPTION.

Также в обработчике исключения можно получить информацию о текущем исключении командой GET STACKED DIAGNOSTICS, которая имеет вид:

GET STACKED DIAGNOSTICS *переменная* { = | := } *элемент* [ , ... ];

Каждый *элемент* представляется ключевым словом, указывающим, какое значение состояния нужно присвоить заданной *переменной* (она должна иметь подходящий тип данных, чтобы принять его). Доступные в настоящее время элементы состояния показаны в [Таблице 42.2](#).



# Exception

Если исключение не устанавливает значение для идентификатора, то возвращается пустая строка.

Пример:

# DECLARE

```
text_var1 text;
```

```
text_var2 text;
```

```
text_var3 text;
```

# BEGIN

-- здесь происходит обработка, которая может вызвать исключение

...

## EXCEPTION WHEN OTHERS THEN

```
GET STACKED DIAGNOSTICS text_var1 = MESSAGE_TEXT,
```

```
text_var2 = PG_EXCEPTION_DETAIL,
```

```
text_var3 = PG_EXCEPTION_HINT;
```

END;

## Получение информации о месте выполнения

Команда `GET DIAGNOSTICS`, ранее описанная в [Подразделе 42.5.5](#), получает информацию о текущем состоянии выполнения кода (тогда как команда `GET STACKED DIAGNOSTICS`, рассмотренная ранее, выдаёт информацию о состоянии выполнения в момент предыдущей ошибки). Её элемент состояния `PG_CONTEXT` позволяет определить текущее место выполнения кода. `PG_CONTEXT` возвращает текст с несколькими строками, описывающий стек вызова. В первой строке отмечается текущая функция и выполняемая в данный момент команда `GET DIAGNOSTICS`, а во второй и последующих строках отмечаются функции выше по стеку вызовов.



The image features a high-angle, aerial view of a dense urban skyline, likely New York City, with numerous skyscrapers and buildings. The entire image is overlaid with a semi-transparent blue and green gradient. A network of thin, light blue lines connects various points across the gradient, creating a digital or technological feel. In the center of the image, the word "Вопросы?" is written in a large, bold, white sans-serif font.

**Вопросы?**



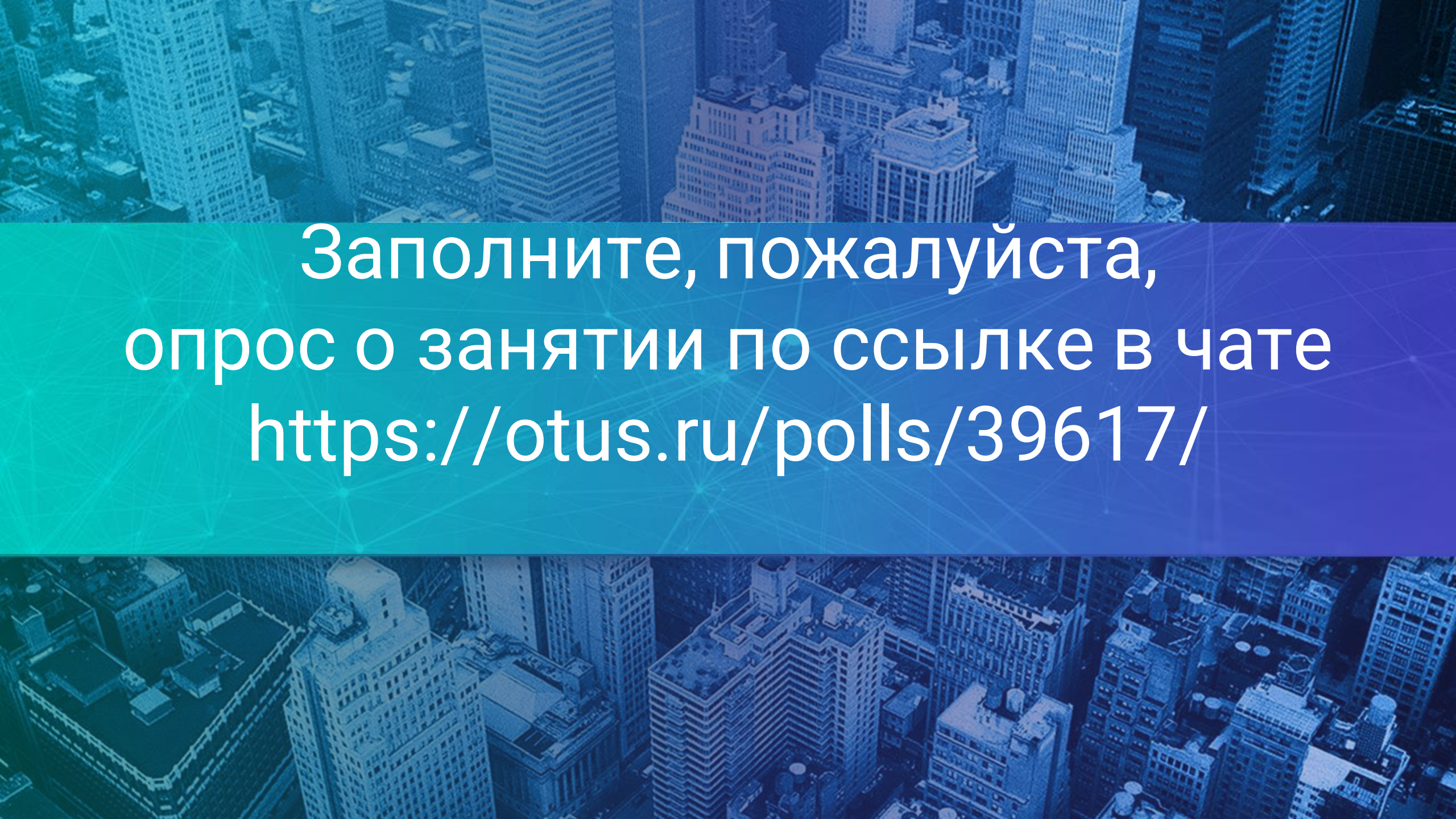
The image features a high-angle, aerial view of a dense urban skyline, likely New York City, with numerous skyscrapers and buildings. The entire image is overlaid with a semi-transparent blue and green gradient. A network of white lines and dots, resembling a digital or social network, is superimposed over the cityscape. The text "Порефлексируем" is centered in the middle of the image in a large, white, sans-serif font.

Порефлексируем



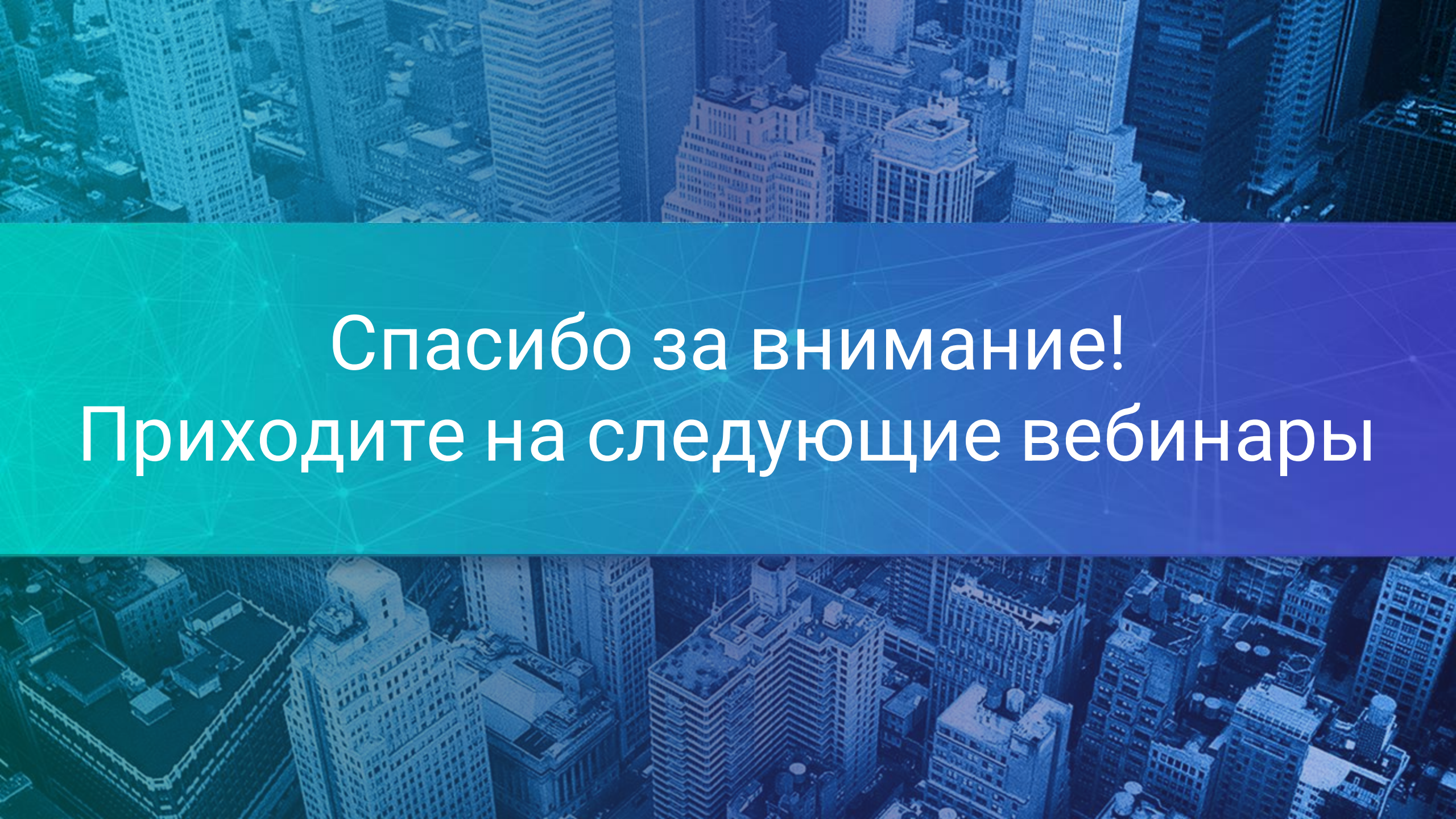
# Вопросы?

- Кто что запомнил?)
- Хватило ли практики?

The background of the image is an aerial photograph of a city, likely New York City, showing a dense cluster of skyscrapers. The entire image is overlaid with a semi-transparent blue layer. On this blue layer, there is a faint, white, geometric network pattern consisting of interconnected lines and dots, resembling a molecular or digital structure. The text is centered in the middle of the image, overlaid on the blue background.

Заполните, пожалуйста,  
опрос о занятии по ссылке в чате  
<https://otus.ru/polls/39617/>



The background of the entire image is an aerial photograph of a city skyline, likely New York City, with numerous skyscrapers. A semi-transparent blue overlay covers the entire image. In the center, there is a horizontal band with a gradient from teal on the left to dark blue on the right. Overlaid on this band is a white network pattern of dots and lines. The text is centered within this band.

Спасибо за внимание!  
Приходите на следующие вебинары