## **Лабораторная работа № 4 Создание хранимых процедур**

**Теоретические сведения**

На практике часто требуется повторять последователь­ность одинаковых запросов. Хранимые процедуры позволяют объединить последова­тельность таких запросов и сохранить их на сервере. После этого клиентам достаточно послать один за­прос на выполнение хранимой процедуры.

Хранимые процедуры обладают следующими преимуществами.

* *Повторное использование кода* – после создания хранимой процедуры ее можно вызывать из любых приложений и SQL-запросов.
* *Сокращение сетевого трафика* – вместо нескольких запросов экономнее послать серверу запрос на выполнение хранимой процедуры и сразу получить ответ.
* *Безопасность* – действия не приведут к нарушению целостности данных, т.к. для выполнения хранимой про­цедуры пользователь должен иметь привилегию.
* *Простота доступа* – хранимые процедуры позволяют инкапсулировать сложный код и оформить его в виде простого вызова.
* *Выполнение бизнес-логики* – хранимые процедуры позволяют перенести код сохранения целостности БД из прикладной программы на сервер БД. Бизнес-логика в виде храни­мых процедур не зависит от языка разработки приложения.

**Создание хранимых процедур**. Реализуется оператором

*CREATE [ OR REPLACE ] FUNCTION*

*name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default\_expr ] [, ...] ] )*

*{ LANGUAGE lang\_name*

*| TRANSFORM { FOR TYPE type\_name } [, ... ]*

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

*| SET configuration\_parameter { TO value | = value | FROM CURRENT }*

*| AS 'definition'*

*| AS 'obj\_file', 'link\_symbol'*

*| sql\_body*

*} ...*

Телом процедуры является составной оператор *begin ... end*, внут­ри которого могут располагаться другие операторы:

*[ label: ] BEGIN*

*statements*

*END [ label ]*

*BEGIN TRANSACTION* (или просто BEGIN) - начинает транзакцию.

*COMMIT* - фиксирует транзакцию, применяя все изменения к базе данных.

*ROLLBACK* - откатывает транзакцию, отменяя все изменения, сделанные в рамках транзакции.

*SAVEPOINT* - устанавливает точку сохранения внутри транзакции, чтобы можно было выполнить частичный откат.

Радует наличие конструкция *AS $$* в PostgreSQL используется для указания начала блока текста хранимой процедуры или функции на языке PL/pgSQL (языке хранимых процедур PostgreSQL). Этот блок текста заключается между двумя символами доллара $$. Это обеспечивает возможность использования многострочного текста без необходимости экранирования специальных символов, что делает код более читабельным.

*CREATE OR REPLACE FUNCTION example\_FUNCTION()*

*LANGUAGE plpgsql*

*AS $$*

*DECLARE*

*-- Здесь идет тело процедуры*

*BEGIN*

*-- Логика процедуры*

*END;*

*$$;*

**Пример кода для версии 15.6:**  
*CREATE OR REPLACE FUNCTION add\_client(*

*p\_address VARCHAR(255),*

*p\_company VARCHAR(255),*

*p\_phone VARCHAR(255),*

*p\_mail VARCHAR(255),*

*p\_name VARCHAR(255),*

*OUT p\_id\_client BIGINT*

*)*

*LANGUAGE plpgsql*

*AS $$*

*BEGIN*

*INSERT INTO Client (cl\_address, cl\_company, cl\_phone, cl\_mail, cl\_name)*

*VALUES (p\_address, p\_company, p\_phone, p\_mail, p\_name)*

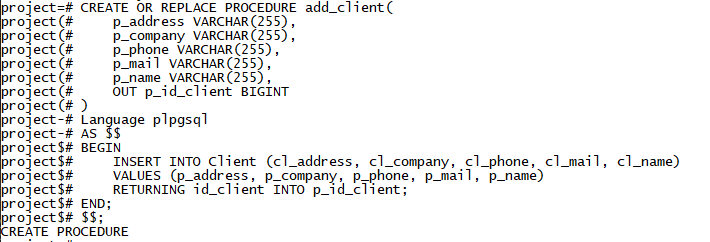
*RETURNING id\_client INTO p\_id\_client;*

*END;*

*$$;*

Оператор, начинающийся с необязательной метки *label* (любое уникальное имя), может заканчиваться выражением *end label*. Внутри составного оператора *begin ... end* может находиться другой составной оператор. Если хранимая процедура содержит один запрос, то составной оператор можно не использовать.

Пример postgres 15:

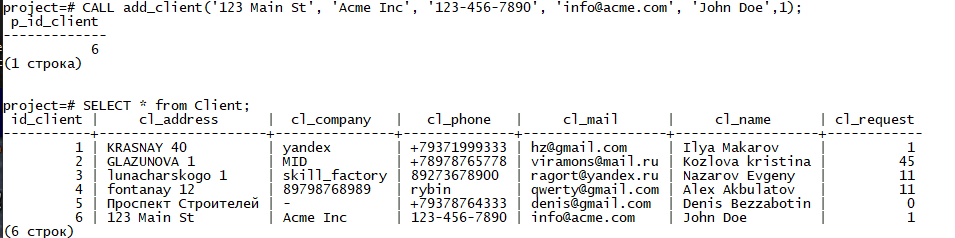


Что­бы вызвать хранимую процедуру, необходимо применить оператор *CALL* в версии выше 11 или *Select* старше 11 версии, после кото­рого помещается имя процедуры и ее параметры в круглых скобках.

Пример postgres 15:

CALL add\_client(*'*123 Main St*', 'Acme Inc', '123-456-7890', 'info@acme.com', 'John Doe',1*);

SELECT \* from Client;



Рекомендуется избегать использования названий хранимых процедур, совпадающих с именами встроенных функций PosgreSQL. В теле хранимой процедуры можно использовать многострочный комментарий, который начинается с последовательности /\* и заканчивается последо­вательностью \*/.

Создадим процедуру *delete\_client()*, которая удаляет определённого клиента в таблице *Client*:

CREATE OR REPLACE FUNCTION *delete\_client(*

*p\_client\_id BIGINT*

*)*

*RETURNS VOID*

*LANGUAGE plpgsql*

*AS $$*

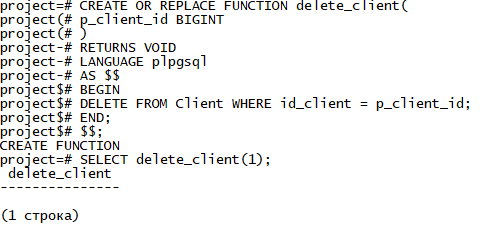
*BEGIN*

*DELETE FROM Client WHERE id\_client = p\_client\_id;*

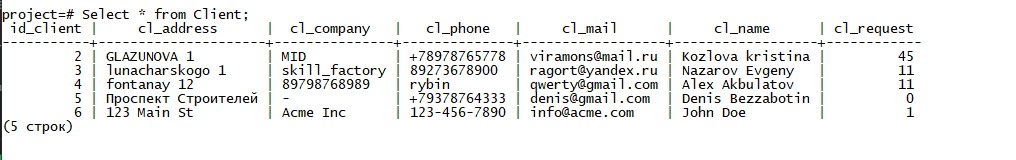
*END;*

*$$;*

*SELECT delete\_client(1);*



Select \* from Client;



Создадим хранимую процедуру *update\_project\_status()*, которая будет изменять состояние проектов на true если дата сдачи проекта уже прошла в зависимости от передаваемой даты.

CREATE OR REPLACE FUNCTION *update\_project\_status (*

*p\_current\_date DATE*

*)*

*RETURNS VOID*

*LANGUAGE plpgsql*

*AS $$*

*BEGIN*

*UPDATE Project*

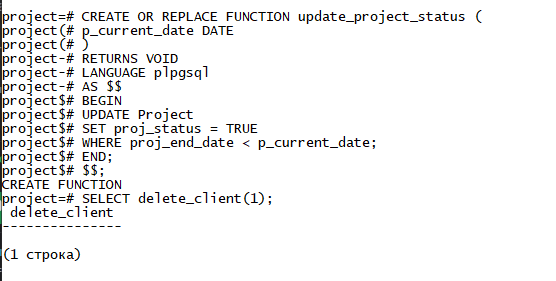
*SET proj\_status = TRUE*

*WHERE proj\_end\_date < p\_current\_date;*

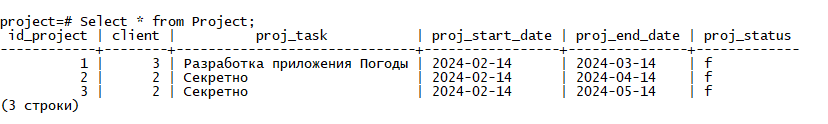
*END;*

*$$;*

*SELECT delete\_client(1);*



Select \* from Project;



**Операторы управления потоком данных**:

* Хранимые процедуры позволяют реализовать сложную логику с помощью операторов ветвления и циклов. Вне хранимых процедур эти операторы применять нельзя. Ветвление программы по условию позволяет реализовать оператор:

*IF лог\_выражение THEN оператор*

*[ELSEIF лог\_выражение THEN оператор] ...*

*[ELSE оператор]*

*END IF ;*

Логическое выражение может принимать два значения:

* 0 (ложь);
* значение, отличное от нуля (истина).

Если логическое выражение истинно, то выполняется опера­тор в блоке *THEN*, иначе выполняется список операторов в блоке *else* (если блок *else* имеется). В логических выражениях можно использовать операторы сравнения ( = , >, >=, <> , <, <= ). Логические выражения можно комбинировать с помощью операторов && (И), а также || (ИЛИ). Если в блоках *if, elseif* и *else* – два или более операторов, необходимо использовать составной оператор *begin ... end*.

Допустим, у нас есть таблица Employee с полем em\_position, которое указывает должность сотрудника. Мы хотим создать процедуру, которая будет выводить сообщение в зависимости от должности сотрудника.

CREATE OR REPLACE FUNCTION *greet\_employee (*

*p\_employee\_position VARCHAR(255)*

*)*

*RETURNS VARCHAR(255)*

*LANGUAGE plpgsql*

*AS $$*

*DECLARE*

*greeting\_message VARCHAR (255);*

*BEGIN*

*IF p\_employee\_position = 'Manager' THEN*

*greeting\_message:= 'Hello,Manager!';*

*ELSEIF p\_employee\_position = 'Developer' THEN*

*greeting\_message:= 'Hello,Developer!';*

*ElSE*

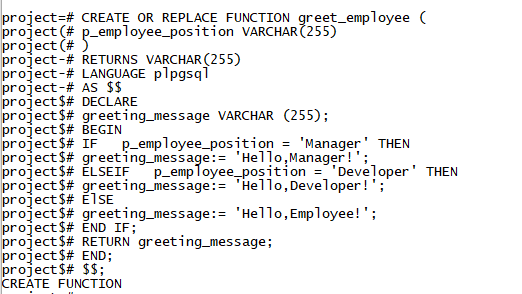
*greeting\_message:= 'Hello,Employee!';*

*END IF;*

*RETURN greeting\_message;*

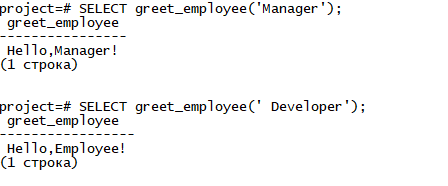
*END;*

*$$;*



SELECT greet\_employee(*'Manager'*);

SELECT greet\_employee(*' Developer'*);



* Множественный выбор позволяет осуществить оператор:

*CASE выражение*

*WHEN значение THEN оператор*

*[WHEN значение THEN оператор] ...*

*[ELSE оператор]*

*END CASE ;*

Выражение сравнивается со значениями. Как только найдено соответствие, выполняется соответствующий оператор. Если соответствия не найдены, выполняется оператор, размещенный после ключевого слова *else* (если оно при­сутствует).

Предположим, у нас есть таблица Employee с полем em\_department, которое указывает на отдел, в котором работает сотрудник. Мы хотим создать процедуру, которая будет возвращать название отдела в зависимости от переданного идентификатора отдела.

CREATE OR REPLACE FUNCTION *get\_department\_name (*

*p\_department\_id BIGINT*

*)*

*RETURNS VARCHAR(255)*

*LANGUAGE plpgsql*

*AS $$*

*DECLARE*

*department\_name VARCHAR (255);*

*BEGIN*

*Department\_name:=*

*CASE p\_department\_id*

*WHEN 1 THEN 'Sales'*

*WHEN 2 THEN 'Marketing'*

*WHEN 3 THEN 'Development'*

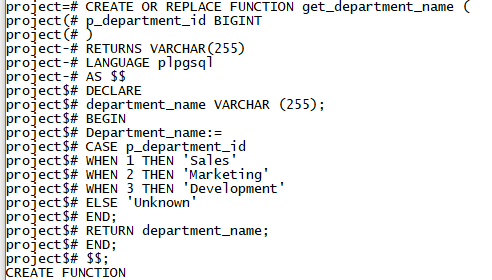
*ELSE 'Unknown'*

*END;*

*RETURN department\_name;*

*END;*

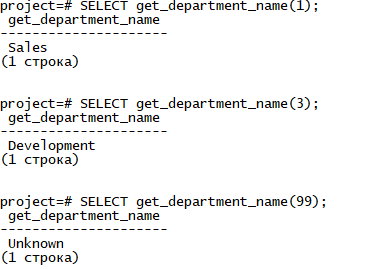
*$$;*



SELECT *get\_department\_name(1);*

SELECT *get\_department\_name(3);*

SELECT *get\_department\_name(99);*



* В PostgreSQL для написания циклов в SQL-запросах и хранимых процедурах используются различные операторы циклов. Вот краткое описание каждого из них:

1. *LOOP*: Этот оператор создает бесконечный цикл, который можно прервать с помощью оператора EXIT или RETURN.

*[<<метка>>]*

*LOOP*

*операторы*

*END LOOP [ метка ];*

CREATE OR REPLACE FUNCTION *loop\_example()*

*RETURNS VOID*

*LANGUAGE plpgsql*

*AS $$*

*DECLARE*

*counter INTEGER:= 1;*

*BEGIN*

*LOOP*

*RAISE NOTICE 'Counter:%', counter;*

*counter:= counter +1;*

*IF counter > 5 THEN*

*EXIT;*

*END IF;*

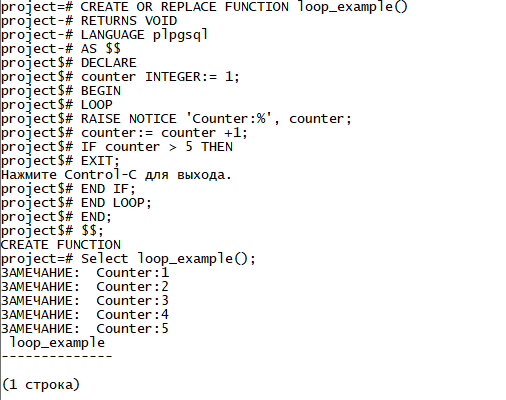
*END LOOP;*

*END;*

*$$;*

*Select loop\_example();*

Пример:

**

1. *WHILE*: Оператор WHILE создает цикл, который выполняется, пока условие истинно.

*[<<метка>>]*

*WHILE логическое-выражение LOOP*

*операторы*

*END LOOP [ метка ];*

Пример:

CREATE OR REPLACE FUNCTION *while\_example()*

*RETURNS VOID*

*LANGUAGE plpgsql*

*AS $$*

*DECLARE*

*counter INTEGER:= 1;*

*BEGIN*

*WHILE counter <= 5 LOOP*

*RAISE NOTICE 'Counter:%', counter;*

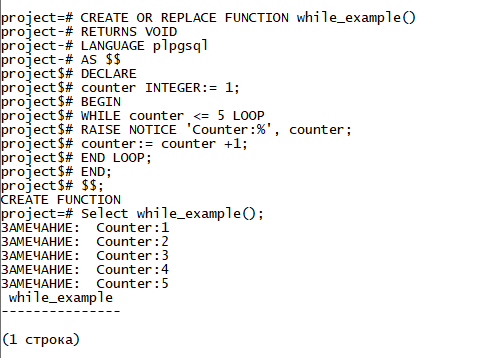
*counter:= counter +1;*

*END LOOP;*

*END;*

*$$;*

*Select while\_example();*



1. *FOR*: Оператор FOR используется для итерации по результатам запроса или набора значений.

*[<<метка>>]*

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

*операторы*

*END LOOP [ метка ];*

CREATE OR REPLACE FUNCTION *for\_example()*

*RETURNS VOID*

*LANGUAGE plpgsql*

*AS $$*

*DECLARE*

*i INTEGER;*

*BEGIN*

*FOR i IN 1..5 LOOP*

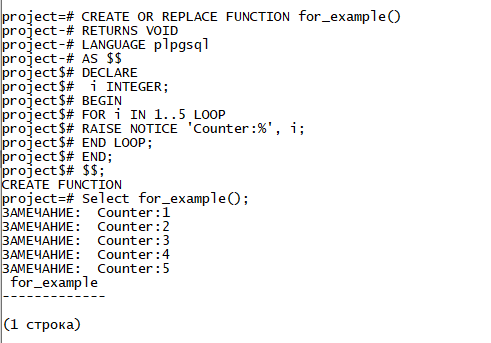
*RAISE NOTICE 'Counter:%', i;*

*END LOOP;*

*END;*

*$$;*

*Select for\_example();*

**

1. Цикл по результатам запроса: Другой вариант FOR позволяет организовать цикл по результатам запроса. Переменная *цель* может быть переменной-кортежем, переменной типа *record* или разделённым запятыми списком скалярных переменных. Переменной *цель* последовательно присваиваются строки результата запроса, и для каждой строки выполняется тело цикла.

*[ <<метка>> ]*

*FOR цель IN запрос LOOP*

*операторы*

*END LOOP [ метка ];*

Предположим, вы хотите создать хранимую процедуру, которая будет выводить имена всех сотрудников в каждом отделе. Мы можем использовать цикл FOR, чтобы итерироваться по отделам и выводить имена сотрудников в каждом из них:

Пример:

CREATE OR REPLACE FUNCTION list\_employees\_by\_department()

RETURNS VOID

*LANGUAGE plpgsql*

*AS $$*

*DECLARE*

department\_id BIGINT;

department\_name VARCHAR(255);

employee\_record Employee;

*BEGIN*

*FOR* department\_id, department\_name IN SELECT id\_department,dep\_task FROM Department *LOOP*

*RAISE NOTICE 'Employees in department % (%):',* department\_name, department\_id *;*

*FOR* employee\_record IN SELECT \* FROM Employee WHERE em\_department = department\_id *LOOP*

*RAISE NOTICE ' - %',* employee\_record.em\_name*;*

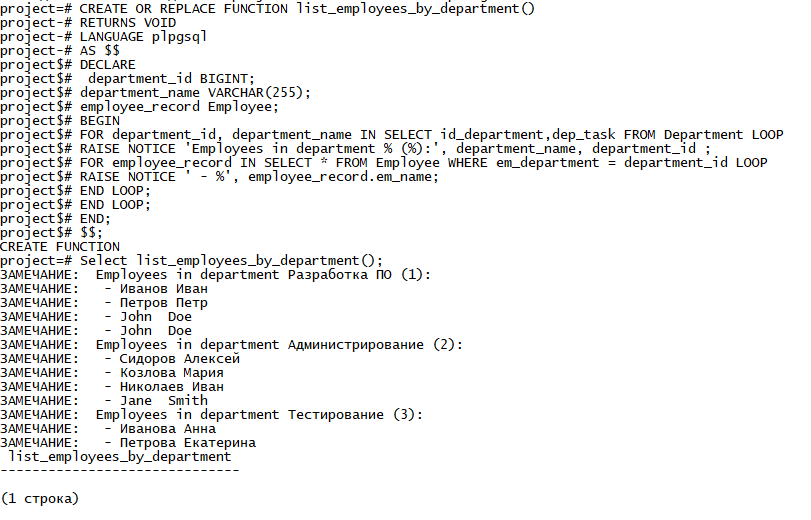
*END LOOP;*

*END LOOP;*

*END;*

*$$;*

Select list\_employees\_by\_department();

**

1. *FOREACH*: Цикл FOREACH очень похож на FOR. Отличие в том, что вместо перебора строк SQL-запроса происходит перебор элементов массива. (В целом, FOREACH предназначен для перебора выражений составного типа. Варианты реализации цикла для работы с прочими составными выражениями помимо массивов могут быть добавлены в будущем.)

*[ <<метка>> ]*

*FOREACH цель [ SLICE число ] IN ARRAY выражение LOOP*

*операторы*

*END LOOP [ метка ];*

CREATE OR REPLACE FUNCTION *foreach\_example()*

*RETURNS VOID*

*LANGUAGE plpgsql*

*AS $$*

*DECLARE*

employee\_record RECORD*;*

*BEGIN*

*FOR employee\_record IN SELECT \* FROM Employee LOOP*

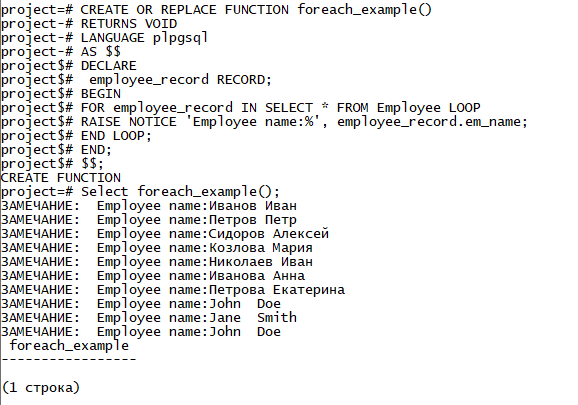
*RAISE NOTICE 'Employee name:%', employee\_record.em\_name;*

*END LOOP;*

*END;*

*$$;*

*Select foreach\_example();*



*\** ***EXIT***: Оператор EXIT используется внутри цикла для выхода из него.

*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.

*\** ***CONTINUE***: Оператор CONTINUE используется для перехода к следующей итерации цикла.

*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.

\* ***RAISE NOTICE*** в PostgreSQL используется для вывода сообщений во время выполнения запросов или хранимых процедур. Он полезен для отладки, мониторинга и сообщения о прогрессе выполнения код

**Удаление хранимых процедур**. Для удаления процедур используется оператор

С 11 версии postgres:

*DROP FUNCTION [IF EXISTS] имя\_процедуры ;*

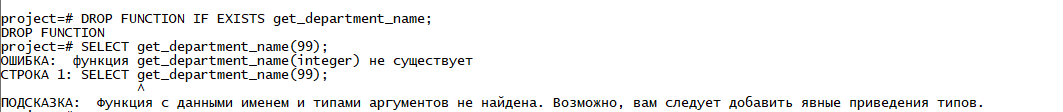
Версии старше postgres 11:

*DROP FUNCTION [IF EXISTS] имя\_процедуры ;*

Пример:

DROP FUNCTION IF EXISTS get\_department\_name;

SELECT get\_department\_name(99);

**

Если удаляемой процедуры с таким именем не существует, оператор возвращает ошибку, которую можно подавить, если использовать необязательное ключевое слово *if exists*.

**Обработчики ошибок**. При выполнении хранимых процедур могут возникать ошибки. PostgreSQL позволяет каждой возникающей в хранимой процедуре ошибке назначить свой обработчик, который в зависимости от ситуации и серьезности ошибки может как прекратить, так и продолжить выполнение процедуры.

Варианты обработчиков:

1. Использование блока EXCEPTION, который позволяет указать код, который будет выполнен в случае возникновения определённой ошибки;
2. Определение собственных обработчиков ошибок с помощью функции DECLARE;
3. Специальные функции для работы с ошибками (например, RAISE и RAISE NOTICE), которые позволяют сгенерировать ошибку с пользовательским сообщением или вывести сообщение об ошибке, но не прерывать выполнение кода.

Пример для EXCEPTION:

CREATE OR REPLACE FUNCTION *add\_project(*

*p\_project\_id BIGINT,*

*p\_client\_id BIGINT,*

*p\_task VARCHAR(255),*

*p\_start\_date DATE,*

*p\_end\_date DATE*

*)*

*RETURNS VOID*

*LANGUAGE plpgsql*

*AS $$*

*BEGIN*

*BEGIN*

INSERT INTO Project (id\_project, client, proj\_task, proj\_start\_date, proj\_end\_date)

VALUES (*p\_project\_id, p\_client\_id,p\_task,p\_start\_date, p\_end\_date*);

RAISE NOTICE *'Project added successfully. ';*

*EXCEPTION*

*WHEN unique\_violation THEN*

RAISE EXCEPTION *'Project with ID % already exists. ',p\_project\_id;*

*WHEN others THEN*

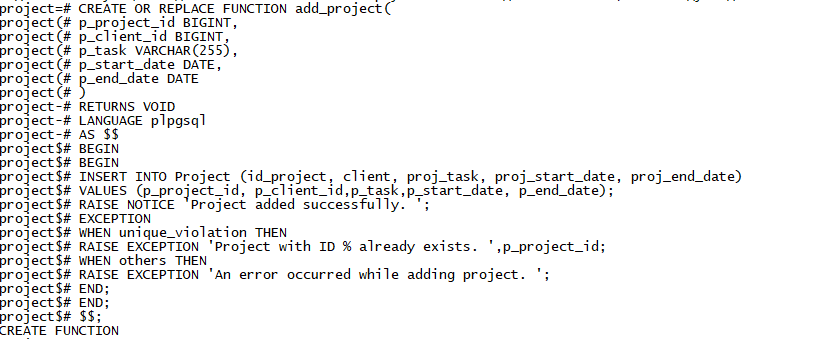
RAISE EXCEPTION *'An error occurred while adding project. ';*

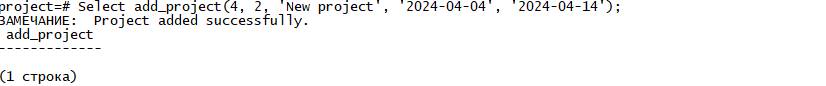
*END;*

*END;*

*$$;*

*Select add\_project(4, 2, 'New project', '2024-04-04', '2024-04-14');*





**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](https://postgrespro.ru/docs/postgresql/14/errcodes-appendix#ERRCODES-TABLE)). 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](https://postgrespro.ru/docs/postgresql/14/plpgsql-control-structures#PLPGSQL-EXCEPTION-DIAGNOSTICS-VALUES):

| **Имя** | **Тип** | **Описание** |
| --- | --- | --- |
| RETURNED\_SQLSTATE | text | код исключения, возвращаемый SQLSTATE |
| COLUMN\_NAME | text | имя столбца, относящегося к исключению |
| CONSTRAINT\_NAME | text | имя ограничения целостности, относящегося к исключению |
| PG\_DATATYPE\_NAME | text | имя типа данных, относящегося к исключению |
| MESSAGE\_TEXT | text | текст основного сообщения исключения |
| TABLE\_NAME | text | имя таблицы, относящейся к исключению |
| SCHEMA\_NAME | text | имя схемы, относящейся к исключению |
| PG\_EXCEPTION\_DETAIL | text | текст детального сообщения исключения (если есть) |
| PG\_EXCEPTION\_HINT | text | текст подсказки к исключению (если есть) |
| PG\_EXCEPTION\_CONTEXT | text | строки текста, описывающие стек вызовов в момент исключения |

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;*

**Курсоры**. Если результирующий запрос возвращает одну запись, поместить результаты в про­межуточные переменные можно с помощью оператора *select ... into ... from*. Однако результирующие таблицы чаще содержат несколько записей, и использование такой конструкции приводит к воз­никновению ошибки 1172: «Результат содержит более чем одну строку».

Избежать ошибки можно, добавив предложение *limit 1*. Однако такая процедура реализует не то поведение, которое ожидает пользователь. Кроме того, существуют ситуации, когда требуется обработать именно многострочную результирующую таблицу.

Например, пусть требуется вернуть записи одной таблицы, отвечающие определенному условию, и на основании этих записей создать новую таблицу. Решить эту задачу можно с помощью курсоров, которые позволяют в цикле просмотреть каждую строку результирующей таблицы запросов. Работа с курсорами похожа на работу с файлами – сначала открытие курсора, затем чтение и после закрытие.

Работа с курсорами происходит по следующему алгоритму:

1. При помощи инструкции *DECLARE* *курсор CURSOR* *FOR* связывается имя курсора с выполняемым запросом.

2. Оператор *open* выполняет запрос, связанный с курсором, и устанавливает курсор перед первой записью результирующей таблицы.

3. Оператор *fetch* помещает курсор на первую запись результирующей таблицы и извлекает данные из записи в локальные переменные хранимой процедуры. По­вторный вызов оператора *fetch* приводит к перемещению курсора к следующей записи, и так до тех пор, пока записи в результирующей таблице не будут исчерпа­ны. Эту операцию удобно осуществлять в цикле.

4. Оператор *close* прекращает доступ к результирующей таблице и ликвидирует связь между курсором и результирующей таблицей.

Давайте рассмотрим пример с использованием курсора для выборки данных из таблицы и последующей обработки этих данных в хранимой процедуре. Предположим, у нас есть таблица Employee с информацией о сотрудниках, и мы хотим создать процедуру, которая будет выводить имена и должности всех сотрудников в алфавитном порядке:

CREATE OR REPLACE FUNCTION *list\_employees()*

*RETURNS VOID*

*LANGUAGE plpgsql*

*AS $$*

*DECLARE*

*Emp\_cursor CURSOR FOR*

*SELECT em\_name, em\_position FROM Employee ORDER BY em\_name;*

*Employee\_record RECORD;*

*BEGIN*

*OPEN emp\_cursor;*

*LOOP*

*FETCH emp\_cursor INTO employee\_record;*

*EXIT WHEN NOT FOUND;*

*RAISE NOTICE 'Name: %,Position: %', employee\_record.em\_name, employee\_record.em\_position;*

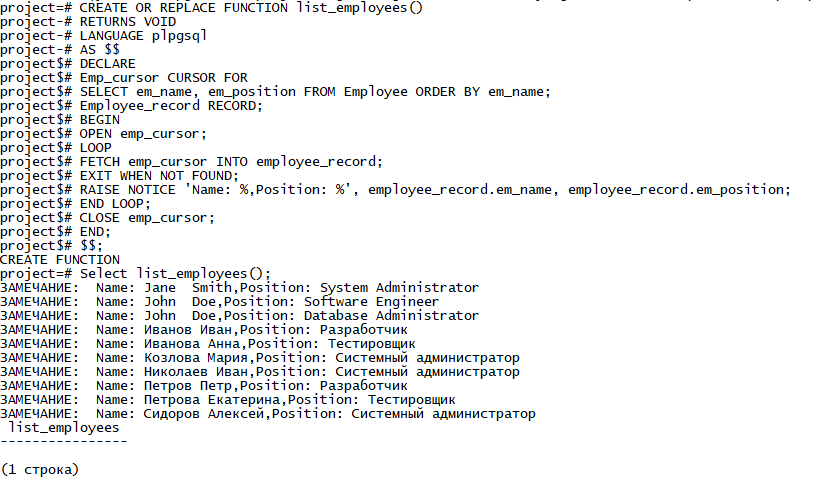
*END LOOP;*

*CLOSE emp\_cursor;*

*END;*

*$$;*

*Select list\_employees();*



**Практическая работа**

При выполнении лабораторной работы необходимо:

* для заданной предметной области написать 6 хранимых процедур и включить их в БД. Каждая процедура должна быть индивидуальной, при решении использовать различные виды операторов управления и обработчики ошибок. Обязательно должны присутствовать две процедуры с обработкой исключений.
* \*Создать одну процедуру с получение информации об ошибки. Взять любой на выбор из таблицы.
* составить отчет по лабораторной работе.

**Пример выполнения работы**

1. Создадим хранимую процедуру, которая меняет местами cl\_phone двух клиентов.

CREATE OR REPLACE FUNCTION *swap\_client\_adresses(*

*p\_client1\_id BIGINT,*

*p\_client2\_id BIGINT)*

*RETURNS VOID*

*LANGUAGE plpgsql*

*AS $$*

*DECLARE*

*temp\_address VARCHAR(255);*

*BEGIN*

*--Получаем адрес первого клиента*

*SELECT cl\_address INTO temp\_address FROM Client WHERE id\_client = p\_client1\_id;*

*--Обновляем адрес первого клиента адресом второго*

*UPDATE Client SET cl\_address= (SELECT cl\_address FROM Client WHERE id\_client = p\_client2\_id) WHERE id\_client = p\_client1\_id;*

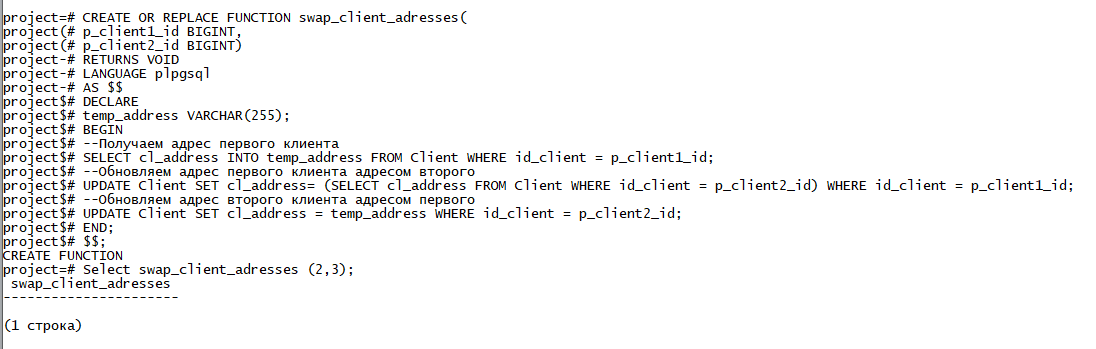
*--Обновляем адрес второго клиента адресом первого*

*UPDATE Client SET cl\_address = temp\_address WHERE id\_client = p\_client2\_id;*

*END;*

*$$;*

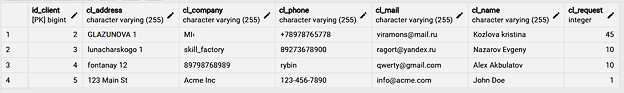
*Select swap\_client\_adresses (2,3);*



Параметр *swap\_client\_addresses* является выходным, его значение равно числу заказов покупателя, код которого записывается во входной параметр *user\_kod.* Процедура считает все строки, где код клиента совпадает с параметром *user\_kod.*

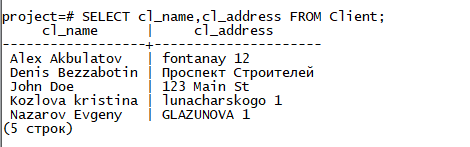
Проверим результат работы после выполнения функции:

до



После:

SELECT cl\_name,cl\_address FROM Client;



2. Создадим функцию, которая будет проверять статус проекта и возвращать информацию о незавершенных проектах, а также подсчитывать количество активных проектов для каждого отдела.

CREATE OR REPLACE FUNCTION *count\_active\_projects\_by\_department()*

*RETURNS TABLE(*

*department\_id BIGINT,*

*active\_projects\_count INT)*

*LANGUAGE plpgsql*

*AS $$*

*DECLARE*

*Department\_record RECORD;*

*active\_projects INT;*

*BEGIN*

*--Инициализируем переменные*

*department\_id :=0;*

*--Создаем временную таблицу для хранения результата*

*CREATE TEMP TABLE temp\_active\_projects\_count(*

*department\_id BIGINT,*

*active\_projects\_count INT*

*) ON COMMIT DROP;*

*FOR department\_record IN SELECT id\_department FROM Department*

*LOOP*

*Department\_id:= department\_record.id\_department;*

*--Считаем количество активных проектов для данного отдела*

*active\_projects := 0;*

*LOOP*

*EXIT WHEN NOT EXISTS (SELECT 1 FROM Project WHERE client IN(SELECT id\_client FROM Client WHERE cl\_request = department\_id)AND proj\_status IS TRUE);*

*active\_projects:= active\_projects +1;*

*END LOOP;*

*-- Добавляем результат во временную таблицу*

*INSERT INTO temp\_active\_projects\_count VALUES (department\_id, active\_projects);*

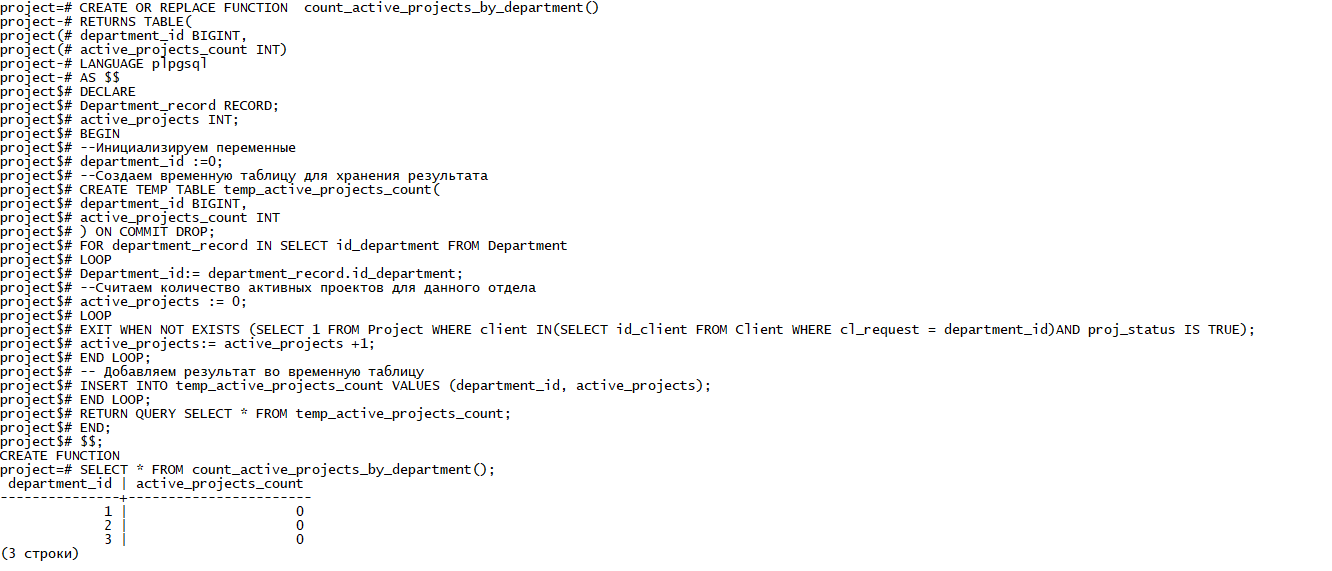
*END LOOP;*

*RETURN QUERY SELECT \* FROM temp\_active\_projects\_count;*

*END;*

*$$;*

*SELECT \* FROM count\_active\_projects\_by\_department();*

**

В этой функции:

* Мы используем цикл FOR для итерации по всем отделам.
* Для каждого отдела мы используем цикл LOOP для подсчета количества активных проектов.
* Мы используем операторы IF и IF ELSE для проверки статуса проекта и увеличения счетчика активных проектов.
* Мы используем оператор CONTINUE для перехода к следующей итерации цикла WHILE, если проект активен.

Эта функция вернет таблицу с двумя столбцами: идентификатор отдела и количество активных проектов для каждого отдела.