Functions

- A PostgreSQL **function** is used to provide a distinct service, and it is often composed of a set of declarations, expressions, and statements.
- PostgreSQL has very rich built-in functions for almost all existing data types.
- In this learning unit, we will focus on user-defined functions.

The PostgreSQL Native Programming Language

- PostgreSQL supports out-of-the-box, user-defined functions, written in C, SQL, and PL/pgSQL.
- There are also three other procedural languages that come with the standard PostgreSQL distribution—PL/Tcl, PL/Python, and PL/Perl.
- However, you will need to create the languages in order to use them, via the `CREATE EXTENSION` PostgreSQL command or the `createlang` utility tool.
- The simplest way to create a language and make it accessible to all databases is to create it in `template1`, directly after the PostgreSQL cluster installation.
- Note that you do not need to perform this step for C, SQL, and PL/pgSQL.

- For beginners, the most convenient languages to use are SQL and PL/pgSQL, since they are supported directly.
- Moreover, they are highly portable, and they do not need special care during the upgrading of the PostgreSQL cluster.
- Creating functions in C is not as easy as creating them in SQL or PL/pgSQL, but since C is a general programming language, you can use it to create very complex functions, in order to handle complex data types, such as images.

Creating a Function in the C Language

- In the following example, we will create a factorial function in the C language.
- This can be used as a template for creating more complex functions.

- You can create a PostgreSQL C function in four steps, as follows:
  1. Install the `postgresql-server-development` library.
  2. Define your function in C, create a `make` file, and compile it as a shared library (**.** `so`).
  3. Specify the location of the shared library that contains your function. The easiest way to do this is to provide the library's absolute path when creating the function or to copy the function-shared library that's created to the PostgreSQL library directory.
  4. Create the function in your database by using the `CREATE FUNCTION` statement.

- To install the PostgreSQL development library, you can use the apt tool, as follows:

```
joshua@joshua-Virtual-Machine:~$ sudo  apt-get install postgresql-server-dev-11
[sudo] password for joshua:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages were automatically installed and are no longer required:
  app-install-data apt-clone archdetect-deb btrfs-tools cryptsetup-bin dmeventd dmraid dpkg-repack gir1.2-timezonemap-1.0 gir1.2-xkl-1.0
  grub-pc-bin kpartx kpartx-boot libdebian-installer4 libdevmapper-event1.02.1 libdmraid1.0.0.rc16 libido3-0.1-0 liblvm2app2.2
  liblvm2cmd2.02 libreadline5 libtimezonemap-data libtimezonemap1 lvm2 python3-icu python3-pam rdate u-boot-tools
```

- In C language development, the `make` tools are often used to compile C code.
- The following is a simple `makefile` to compile the factorial function.
- `pg_config` is used to provide information about the installed version of PostgreSQL:

```
joshua@joshua-Virtual-Machine:~$ cd Documents/postgres/usr_function/c/fact/
joshua@joshua-Virtual-Machine:~/Documents/postgres/usr_function/c/fact$ ls
fact.c  makefile
joshua@joshua-Virtual-Machine:~/Documents/postgres/usr_function/c/fact$ cat makefile
MODULES = fact

PG_CONFIG = pg_config
PGXS = $(shell $(PG_CONFIG) --pgxs)
INCLUDEDIR = $(shell $(PG_CONFIG) --includedir-server)
include $(PGXS)

fact.so: fact.o
        cc -shared -o fact.so fact.o

fact.o: fact.c
        cc -o fact.o -c fact.c $(CFLAGS) -I$(INCLUDEDIR)
```

- The source code of the factorial fact for the abbreviated C function is as follows:

```
joshua@joshua-Virtual-Machine:~/Documents/postgres/usr_function/c/fact$ cat fact.c
#include "postgres.h"
#include "fmgr.h"
#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif
Datum fact(PG_FUNCTION_ARGS);
PG_FUNCTION_INFO_V1(fact);
Datum
fact(PG_FUNCTION_ARGS) {
int32 fact = PG_GETARG_INT32(0);
int32 count = 1, result = 1;
for (count = 1; count <= fact; count++)
    result = result * count;
PG_RETURN_INT32(result);
}
```

- You can compile the code as follows:

```
joshua@joshua-Virtual-Machine:~/Documents/postgres/usr_function/c/fact$ make -f makefile
cc -o fact.o -c fact.c -Wall -Wmissing-prototypes -Wpointer-arith -Wdeclaration-after-statement -Wendif-labels -Wmissing-format-attribute -Wfo
rmat-security -fno-strict-aliasing -fwrapv -fexcess-precision=standard -Wno-format-truncation -g -g -O2 -fstack-protector-strong -Wformat -Wer
ror=format-security -fno-omit-frame-pointer -fPIC -I/usr/include/postgresql/11/server
cc -shared -o fact.so fact.o
/usr/bin/clang-6.0 -Wno-ignored-attributes -fno-strict-aliasing -fwrapv -O2  -I. -I./ -I/usr/include/postgresql/11/server -I/usr/include/postg
resql/internal  -Wdate-time -D_FORTIFY_SOURCE=2 -D_GNU_SOURCE -I/usr/include/libxml2  -I/usr/include/mit-krb5 -flto=thin -emit-llvm -c -o fact
.bc fact.c
```

- Now, you need to copy the file, and this requires `sudo` privileges:

```
joshua@joshua-Virtual-Machine:~/Documents/postgres/usr_function/c/fact$ sudo cp fact.so $(pg_config --pkglibdir)/
```

- Finally, as a `postgres` user, create the function in the template library and test it:

```
joshua@joshua-Virtual-Machine:~/Documents/postgres/usr_function/c/fact$ sudo -u postgres -s
postgres@joshua-Virtual-Machine:~/Documents/postgres/usr_function/c/fact$ psql -d template1 -c "CREATE FUNCTION fact(INTEGER) RETURNS INTEGER
AS 'fact', 'fact' LANGUAGE C STRICT;"
CREATE FUNCTION
postgres@joshua-Virtual-Machine:~/Documents/postgres/usr_function/c/fact$ psql -d template1 -c "SELECT fact(5);"
 fact
------
  120
(1 row)
```

- Writing C functions is quite complicated when compared to SQL and PL/pgSQL functions.
- They might even cause some complications when upgrading the database if they are not well maintained.

- The body of the SQL function can be composed of several SQL statements; the result of the last SQL statement determines the function return type.

```
joshua@joshua-Virtual-Machine:~/Documents/postgres/usr_function/c/fact$ sudo -u postgres psql
[sudo] password for joshua:
psql (11.1 (Ubuntu 11.1-3.pgdg18.04+1))
Type "help" for help.

postgres=# \c car_portal
You are now connected to database "car_portal" as user "postgres".
car_portal=# set role car_portal_app;
SET
car_portal=> CREATE OR REPLACE FUNCTION is_updatable_view (text) RETURNS BOOLEAN AS $$
car_portal$>     SELECT is_insertable_into='YES'
car_portal$>     FROM information_schema.tables
car_portal$>     WHERE table_type = 'VIEW' AND table_name = $1
car_portal$> $$ LANGUAGE SQL;
CREATE FUNCTION
```

```
car_portal=> select is_updatable_view('user_account');
 is_updatable_view
-------------------
 t
(1 row)
```

- An SQL PostgreSQL function cannot be used to construct dynamic SQL statements, since the function argument can only be used to substitute data values and not identifiers.
- The following snippet is not valid in an SQL function:

```
car_portal=> CREATE FUNCTION drop_table (text) RETURNS VOID AS $$
car_portal$>  DROP TABLE $1;
car_portal$> $$ LANGUAGE SQL;
ERROR:  syntax error at or near "$1"
LINE 2:  DROP TABLE $1;
                    ^
```

- The PL/pgSQL language is full-fledged and is the preferable choice for use on a daily basis.
- It can contain a variable declaration, conditional and looping constructs, exception trapping, and so on.
- The following function returns the factorial of an integer:

```
car_portal=> CREATE OR REPLACE FUNCTION fact(fact INT) RETURNS INT AS
car_portal-> $$
car_portal$> DECLARE
car_portal$>     count INT = 1;
car_portal$>     result INT = 1;
car_portal$> BEGIN
car_portal$>     FOR count IN 1..fact LOOP
car_portal$>         result = result* count;
car_portal$>     END LOOP;
car_portal$>     RETURN result;
car_portal$> END;
car_portal$> $$ LANGUAGE plpgsql;
CREATE FUNCTION
car_portal=> select fact(5);
 fact
------
  120
(1 row)
```

Function Usage

- PostgreSQL can be used in several different scenarios.
- For example, some developers use functions as an abstract interface with higher programming languages to hide the data model.
- Additionally, functions have several other uses, as follows:
  - Performing complex logic that's difficult to perform with SQL
  - Performing actions before or after the execution of an SQL statement, via the trigger system
  - Cleaning SQL code by reusing the common code and bundling the SQL code in modules
  - Automating common tasks related to a database by utilizing dynamic SQL

Function Dependencies

- When using PostgreSQL functions, you need to be careful not to end up with dangling functions, since the dependency between functions is not well maintained in the PostgreSQL system catalog.
- The following example shows how you can end up with a dangling function:

```
car_portal=> CREATE OR REPLACE FUNCTION test_dep (INT) RETURNS INT AS $$
car_portal$> BEGIN
car_portal$>     RETURN $1;
car_portal$> END;
car_portal$> $$
car_portal-> LANGUAGE plpgsql;
CREATE FUNCTION
car_portal=> CREATE OR REPLACE FUNCTION test_dep_2(INT) RETURNS INT AS $$
car_portal$> BEGIN
car_portal$>     RETURN test_dep($1);
car_portal$> END;
car_portal$> $$
car_portal-> LANGUAGE plpgsql;
CREATE FUNCTION
car_portal=> DROP FUNCTION test_dep(int);
DROP FUNCTION
```

- The `test_dep_2` function is dangling.
- When this function is invoked, an error will be raised, as follows:

```
car_portal=> SELECT test_dep_2 (5);
ERROR:  function test_dep(integer) does not exist
LINE 1: SELECT test_dep($1)
               ^
HINT:  No function matches the given name and argument types. You might need to add explicit type casts.
QUERY:  SELECT test_dep($1)
CONTEXT:  PL/pgSQL function test_dep_2(integer) line 3 at RETURN
```

PostgreSQL Function Categories

- When you are creating a function, it is marked as volatile by default if the volatility classification is not specified.
- If the created function is not volatile, it is important to mark it as stable or immutable, because this will help the optimizer to generate the optimal execution plans.

- PostgreSQL functions can fall into one of the following volatility classifications:
  - **Volatile**: A volatile function can return different results for successive calls, even if the function argument didn't change; it can also change the data in the database. The `random()` function is a volatile function.
  - **Stable and immutable**: These functions cannot modify the database, and they are guaranteed to return the same results for the same arguments. A `stable` function provides this guarantee within the statement scope, while an `immutable` function provides this guarantee globally, without any scope.

- For example, the `random()` function is volatile, since it will give a different result for each call.
- The `round()` function is immutable, because it will always give the same result for the same argument.

- The `now()` function is stable, since it will always give the same result within the statement or transaction, as shown in the following code snippet:

```
car_portal=> BEGIN;
BEGIN
car_portal=> SELECT now();
              now
-------------------------------
 2019-03-06 10:59:50.569213+08
(1 row)

car_portal=> SELECT 'Some time has passed', now();
       ?column?       |              now
----------------------+-------------------------------
 Some time has passed | 2019-03-06 10:59:50.569213+08
(1 row)

car_portal=> ROLLBACK;
ROLLBACK
```

PostgreSQL Anonymous Functions

- PostgreSQL provides the `DO` statement, which can be used to execute anonymous code blocks.
- The `DO` statement reduces the need to create shell scripts for administration purposes.
- However, note that all PostgreSQL functions are transactional; so, if you would like to create indexes, for example, shell scripting is a better alternative, or procedures, which are introduced in PostgreSQL 11.

- let's assume that we would like to have another user who can only perform `select` statements.
- You can create a role, as follows:

```
joshua@joshua-Virtual-Machine:~/Documents/postgres/usr_function/c/fact$ sudo -u postgres psql
[sudo] password for joshua:
psql (11.1 (Ubuntu 11.1-3.pgdg18.04+1))
Type "help" for help.

postgres=# \c car_portal
You are now connected to database "car_portal" as user "postgres".
car_portal=# CREATE user select_only;
CREATE ROLE
car_portal=# GRANT CONNECT ON DATABASE car_portal TO select_only;
GRANT
car_portal=# GRANT USAGE ON SCHEMA car_portal_app TO select_only;
GRANT
```

- Now, you need to grant select permission on each table for the newly created role.
- This can be done using dynamic SQL, as follows:

```
car_portal=# DO $$
car_portal$#  DECLARE r record;
car_portal$# BEGIN
car_portal$#  FOR r IN SELECT table_schema, table_name FROM information_schema.tables WHERE table_schema = 'car_portal_app' LOOP
car_portal$#   EXECUTE 'GRANT SELECT ON ' || quote_ident(r.table_schema) || '.'|| quote_ident(r.table_name) || ' TO select_only';
car_portal$#  END LOOP;
car_portal$# END$$;
DO
car_portal=# set role select_only;
SET
car_portal=> select * from car_portal_app.account limit 1;
 account_id | first_name | last_name |      email      |             password
------------+------------+-----------+-----------------+----------------------------------
          1 | James      | Butt      | jbutt@gmail.com | 1b9ef408e82e38346e6ebebf2dcc5ece
(1 row)
```