User, User, Who Is the User?

BRUCE MOMJIAN

This talk the many options available for Postgres user management.

https://momjian.us/presentations

Last updated: April 2025

The Cluster's Original Superuser

When you create a PostgreSQL cluster with *initdb,* a single all-powerful superuser, called the bootstrap user, is created. Typically this user is called *postgres:*

```
$ su - postgres
$ mkdir data
$ chmod 0700 data
$ initdb data
The files belonging to this database system will be owned by user "postgres".
This user must also own the server process.
…
Success. You can now start the database server using:

  pg_ctl -D data -l logfile start

$ pg_ctl -l server.log -D data start
waiting for server to start.... done
server started
```

The Cluster's Original Superuser

```
$ psql postgres
psql (17devel)
Type "help" for help.
postgres=> SELECT CURRENT_USER;
 current_user
--------------
 postgres

postgres=> \du
                             List of roles
 Role name |                         Attributes
-----------+------------------------------------------------------------
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS
```

It Can Be any Operating System User

```
$ su - bruce
$ mkdir data
$ chmod 0700 data
$ initdb data
The files belonging to this database system will be owned by user "bruce".
This user must also own the server process.
…
Success. You can now start the database server using:

   pg_ctl -D data -l logfile start

$ pg_ctl -l server.log -D data start
waiting for server to start.... done
server started
```

It Can Be any Operating System User

```
$ psql postgres   # notice the database is still called "postgres"
SELECT CURRENT_USER;
 current_user
--------------
 bruce

\du
                             List of roles
 Role name |                         Attributes
-----------+------------------------------------------------------------
 bruce     | Superuser, Create role, Create DB, Replication, Bypass RLS
```

Users, Groups, Roles

While CREATE USER and CREATE GROUP commands exist, Postgres follows the SQL standard and focuses on roles (and CREATE ROLE) for user management. Roles with LOGIN permissions are considered users, since they can log in, and roles designated as NOLOGIN (the default) can be considered groups, though LOGIN roles can also function as groups, e.g.:

```
-- CREATE USER defaults to LOGIN ability, while the others are NOLOGIN
CREATE USER demo_user;
CREATE GROUP demo_group;
CREATE ROLE demo_role;

\du demo_*
       List of roles
 Role name  |  Attributes
------------+--------------
 demo_group | Cannot login
 demo_role  | Cannot login
 demo_user  |
```

Roles, not Rolls

https://www.flickr.com/photos/itshomemade/

CURRENT_USER and CURRENT_ROLE

```
\c - demo_user

SELECT CURRENT_USER;
 current_user
--------------
 demo_user

SELECT CURRENT_ROLE;
 current_role
--------------
 demo_user
```

Creating More Roles

```
\h create role
Command:     CREATE ROLE
Description: define a new database role
Syntax:
CREATE ROLE name [ [ WITH ] option [ ... ] ]
where option can be:
      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED ] PASSWORD 'password' | PASSWORD NULL
    | VALID UNTIL 'timestamp'
    | IN ROLE role_name [, ...]
    | ROLE role_name [, ...]
    | ADMIN role_name [, ...]
    | SYSID uid
URL: https://www.postgresql.org/docs/current/sql-createrole.html
```

This presentation covers the red items.

Why Multiple Roles?

Relational databases allow multiple users to access the database at the same time with minimal need for user adjustments, e.g., SELECT FOR UPDATE/SHARE, transaction isolation levels, DDL. While all users could log in with the same user name, there are advantages of logging in using assigned names. There are also advantages of creating non-login roles for permission managment:

1. Object ownership simplifies accounting
2. Database ownership allows control of schemas and extensions
3. Connection control
4. Simplified monitoring of session activity
5. User settings
6. User hierarchies
7. Object access control
8. Predefined roles

These articles by Ryan Booz also has helpful information: https://www.red-gate.com/simple-talk/databases/postgresql/postgresql-basics-roles-and-privileges/, https://www.red-gate.com/simple-talk/homepage/postgresql-basics-object-ownership-and-default-privileges/.

1. Object Ownership Simplifies Accounting

```
\c - postgres
DO $$
DECLARE i iNTEGER;
BEGIN
        FOR i IN 1..15 LOOP
                EXECUTE format('CREATE TABLE single_owner_%s (x INTEGER)',
                                right('0' || i, 2));
        END LOOP;
END;
$$ LANGUAGE plpgsql;
```

All the queries used in this presentation are available at https://momjian.us/main/writings/pgsql/user.sql; execute with caution since this script creates global objects.

Object Ownership Simplifies Accounting

```
\dt single_owner_*
              List of relations
 Schema |       Name        | Type  |  Owner
--------+-------------------+-------+----------
 public | single_owner_01 | table | postgres
 public | single_owner_02 | table | postgres
 public | single_owner_03 | table | postgres
 public | single_owner_04 | table | postgres
 public | single_owner_05 | table | postgres
 public | single_owner_06 | table | postgres
 public | single_owner_07 | table | postgres
 public | single_owner_08 | table | postgres
 public | single_owner_09 | table | postgres
 public | single_owner_10 | table | postgres
 public | single_owner_11 | table | postgres
 public | single_owner_12 | table | postgres
 public | single_owner_13 | table | postgres
 public | single_owner_14 | table | postgres
 public | single_owner_15 | table | postgres
```

Object Ownership Simplifies Accounting

```
DO $$
DECLARE i iNTEGER;
BEGIN
        FOR i IN 1..15 LOOP
                EXECUTE format('CREATE USER user_%s',
                                right('0' || i, 2));
                EXECUTE format('CREATE TABLE multi_owner_%s (x INTEGER)',
                                right('0' || i, 2));
                EXECUTE format('ALTER TABLE multi_owner_%s OWNER TO user_%s',
                                right('0' || i, 2), right('0' || i, 2));
        END LOOP;
END;
$$ LANGUAGE plpgsql;
```

The table owner can set permissions on the table and drop it.

Object Ownership Simplifies Accounting

```
\dt multi_owner_*
            List of relations
 Schema |      Name       | Type  |  Owner
--------+-----------------+-------+---------
 public | multi_owner_01 | table | user_01
 public | multi_owner_02 | table | user_02
 public | multi_owner_03 | table | user_03
 public | multi_owner_04 | table | user_04
 public | multi_owner_05 | table | user_05
 public | multi_owner_06 | table | user_06
 public | multi_owner_07 | table | user_07
 public | multi_owner_08 | table | user_08
 public | multi_owner_09 | table | user_09
 public | multi_owner_10 | table | user_10
 public | multi_owner_11 | table | user_11
 public | multi_owner_12 | table | user_12
 public | multi_owner_13 | table | user_13
 public | multi_owner_14 | table | user_14
 public | multi_owner_15 | table | user_15
```

2. Database Ownership Allows Control of Schemas and Extensions

```
CREATE DATABASE db_01 OWNER user_01;

\l db_01
                                List of databases
 Name  |  Owner  | Encoding | Locale Provider |   Collate   …
-------+---------+----------+-----------------+-------------…
 db_01 | user_01 | UTF8     | libc            | en_US.UTF-8 …

\c - user_01

GRANT ALL ON SCHEMA public TO PUBLIC;
WARNING:  no privileges were granted for "public"

CREATE EXTENSION fuzzystrmatch;
ERROR:  permission denied to create extension "fuzzystrmatch"
HINT:  Must have CREATE privilege on current database to create this extension.
```

Database Ownership Allows Control of Schemas and Extensions

```
\c db_01 user_01

SELECT current_user;
 current_user
--------------
 user_01

-- Starting in PostgreSQL 15, by default the public schema only allows access
-- by the database owner.
GRANT ALL ON SCHEMA public TO PUBLIC;

CREATE EXTENSION fuzzystrmatch;
```

3. Connection Control

pg_hba.conf

```
# TYPE  DATABASE        USER            ADDRESS                 METHOD

# For "sameuser", the database and user names must match
host    sameuser        all             172.20.0.0/16           scram-sha-256
host    all             user_01         172.20.4.18             scram-sha-256
```

```
\c - postgres

ALTER USER user_01 CONNECTION LIMIT 8;
```

4. Simplified Monitoring of Session Activity

```
\! psql --username user_01 -c 'SELECT pg_sleep(3)' test &
\! psql --username user_02 -c 'SELECT pg_sleep(3)' test &
\! psql --username user_03 -c 'SELECT pg_sleep(3)' test &

-- give background processes time to start
SELECT pg_sleep(1);

SELECT usename, current_timestamp - query_start, query
FROM pg_stat_activity
WHERE usename IS NOT NULL AND
      state = 'active' AND
      pid != pg_backend_pid();
```

Clients can also set the server variable `application_name` to improve session monitoring; `application_name` can also be changed during sessions to reflect session activity; see https://momjian.us/main/blogs/pgblog/2017.html#March_13_2017.

Simplified Monitoring of Session Activity

```
 usename |    ?column?     |       query
---------+-----------------+--------------------
 user_01 | 00:00:04.455078 | SELECT pg_sleep(3)
 user_02 | 00:00:02.893516 | SELECT pg_sleep(3)
 user_03 | 00:00:01.523733 | SELECT pg_sleep(3)
```

5. User Settings

```
ALTER USER user_01 SET work_mem = '6MB';
ALTER USER user_02 SET work_mem = '8MB';
ALTER USER user_03 SET work_mem = '10MB';

\c - user_01
SHOW work_mem;
 work_mem
----------
 6MB

\c - user_02
SHOW work_mem;
 work_mem
----------
 8MB

\c - user_03
SHOW work_mem;
 work_mem
----------
```

6. User Hierarchies

```
\c - postgres

CREATE ROLE user_01a IN ROLE user_01 LOGIN;

SELECT rolname, roleid::regrole AS "Is member of role"
FROM  pg_roles, pg_auth_members
WHERE pg_roles.oid = member AND
      rolname NOT LIKE 'pg_%'
ORDER BY 1, 2;
  rolname   |  Is member of role
------------+---------------------
 user_01a   | user_01
```

First Value of Membership: SET ROLE

By default, roles can become member roles:

```
\c - user_01a

SELECT current_user;
 current_user
--------------
 user_01a

SET ROLE user_01;

SELECT current_user;
 current_user
--------------
 user_01
```

Second Value of Membership: INHERITANCE

By default, roles can also modify member-owned tables as though they were the owners of the tables, without having to use SET ROLE to become member roles:

```
GRANT ALL ON SCHEMA public TO PUBLIC;

\c - user_01
CREATE TABLE drop_test(x INTEGER);

\c - user_01a
DROP TABLE drop_test;
```

Membership Without Inheritance

When a role is added as a member with NOINHERIT, SET ROLE must be used to obtain the privileges of member roles:

```
\c - postgres

CREATE ROLE user_01b IN ROLE user_01 LOGIN NOINHERIT;

\c - user_01
CREATE TABLE drop_test1 (x INTEGER);

\c - user_01b
DROP TABLE drop_test1;
ERROR:  must be owner of table drop_test1

SET ROLE user_01;
DROP TABLE drop_test1;
```

Prior to PostgreSQL 16, inheritance was only a role attribute, meaning that roles it was a member of were either all inherited or not inherited; see http://rhaas.blogspot.com/2023/01/surviving-without-superuser-coming-to.html for a summary of Postgres 16 changes.

Membership Without Inheritance

```
SELECT rolname, roleid::regrole AS "Is member of role", inherit_option
FROM  pg_roles, pg_auth_members
WHERE member = pg_roles.oid AND
      rolname NOT LIKE 'pg_%'
ORDER BY 1, 2;
  rolname   |  Is member of role   | inherit_option
------------+---------------------+----------------
 user_01a   | user_01             | t
 user_01b   | user_01             | f
```

Removing Inheritance After Membership

Inheritance can be removed after the role is created:

```
\c - postgres

REVOKE INHERIT OPTION FOR user_01 FROM user_01a;

\c - user_01
CREATE TABLE drop_test2 (x INTEGER);

\c - user_01a
DROP TABLE drop_test2;
ERROR:  must be owner of table drop_test2

SET ROLE user_01;
DROP TABLE drop_test2;
```

Removing Inheritance After Membership

```
SELECT rolname, roleid::regrole AS "Is member of role", inherit_option
FROM  pg_roles, pg_auth_members
WHERE member = pg_roles.oid AND
      rolname NOT LIKE 'pg_%'
ORDER BY 1, 2;
  rolname   |  Is member of role   | inherit_option
------------+---------------------+----------------
 user_01a   | user_01             | f
 user_01b   | user_01             | f
```

Membership Without SET ROLE

This role now has no SET ROLE permission:

```
\c - postgres

REVOKE SET OPTION FOR user_01 FROM user_01a;

\c - user_01
CREATE TABLE drop_test3 (x INTEGER);

\c - user_01a
DROP TABLE drop_test3;
ERROR:  must be owner of table drop_test3

SET ROLE user_01;
ERROR:  permission denied to set role "user_01"
```

Prior to PostgreSQL 16, SET ROLE could not be disabled for members.

Membership Without SET ROLE

```
SELECT rolname, roleid::regrole AS "Is member of role", inherit_option, set_option
FROM  pg_roles, pg_auth_members
WHERE member = pg_roles.oid AND
      rolname NOT LIKE 'pg_%'
ORDER BY 1, 2;
 rolname  | Is member of role | inherit_option | set_option
----------+-------------------+----------------+------------
 user_01a | user_01           | f              | f
 user_01b | user_01           | f              | t
```

Membership Can Be Added After Role Creation

```
\c - postgres

GRANT user_01 TO user_02;
GRANT user_01 TO user_03 WITH INHERIT FALSE;

SELECT rolname, roleid::regrole AS "Is member of role", inherit_option, set_option
FROM  pg_roles, pg_auth_members
WHERE member = pg_roles.oid AND
      rolname NOT LIKE 'pg_%'
ORDER BY 1, 2;
 rolname  | Is member of role | inherit_option | set_option
----------+-------------------+----------------+------------
 user_01a | user_01           | f              | f
 user_01b | user_01           | f              | t
 user_02  | user_01           | t              | t
 user_03  | user_01           | f              | t
```

Membership Can Be Removed

```
REVOKE user_01 FROM user_03;

SELECT rolname, roleid::regrole AS "Is member of role", inherit_option, set_option
FROM  pg_roles, pg_auth_members
WHERE member = pg_roles.oid AND
      rolname NOT LIKE 'pg_%'
ORDER BY 1, 2;
 rolname   | Is member of role | inherit_option | set_option
-----------+-------------------+----------------+------------
 user_01a | user_01           | f              | f
 user_01b | user_01           | f              | t
 user_02  | user_01           | t              | t
```

ADMIN Allows Membership Control

The ADMIN membership attribute allows membership control of other roles:

```
\c - postgres

GRANT user_01 TO user_01a WITH ADMIN TRUE;

\c - user_01a
GRANT user_01 TO user_04;

SELECT rolname, roleid::regrole AS "Is member of role", inherit_option, set_option, admin_opti
FROM  pg_roles, pg_auth_members
WHERE member = pg_roles.oid AND
      rolname NOT LIKE 'pg_%'
ORDER BY 1, 2;
 rolname  | Is member of role | inherit_option | set_option | admin_option
----------+-------------------+----------------+------------+--------------
 user_01a | user_01           | f              | f          | t
 user_01b | user_01           | f              | t          | f
 user_02  | user_01           | t              | t          | f
 user_04  | user_01           | t              | t          | f
```

ADMIN Allows Membership Control

```
-- This was added to psql in PostgreSQL 16.
\drg
                List of role grants
 Role name | Member of |   Options    | Grantor
-----------+-----------+--------------+----------
 user_01a  | user_01   | ADMIN        | postgres
 user_01b  | user_01   | SET          | postgres
 user_02   | user_01   | INHERIT, SET | postgres
 user_04   | user_01   | INHERIT, SET | user_01a
```

GRANT Allows Membership Attribute Changes

```
-- This was added in PostgreSQL 16.
\c - postgres

\drg user_01a
            List of role grants
 Role name | Member of | Options | Grantor
-----------+-----------+---------+----------
 user_01a  | user_01   | ADMIN   | postgres

GRANT user_01 TO user_01a WITH INHERIT TRUE, SET TRUE, ADMIN FALSE;

\drg user_01a
               List of role grants
 Role name | Member of |   Options    | Grantor
-----------+-----------+--------------+----------
 user_01a  | user_01   | INHERIT, SET | postgres
```

Multiple Membership

This role will have three members, and will be a member of three roles:

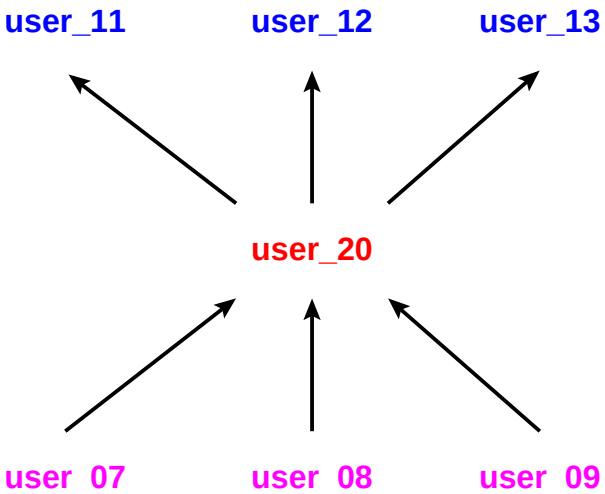
```
CREATE ROLE user_20 ROLE user_07, user_08, user_09 IN ROLE user_11, user_12, user_13;

SELECT rolname, roleid::regrole AS "Is member of role", inherit_option,
       set_option, admin_option
FROM  pg_roles, pg_auth_members
WHERE member = pg_roles.oid AND
      rolname NOT LIKE 'pg_%'
ORDER BY 1, 2;
```

Multiple Membership

rolname	Is member of role	inherit_option	set_option	admin_option
user_01a	user_01	f	f	t
user_01b	user_01	f	t	f
user_02	user_01	t	t	f
user_04	user_01	t	t	f
user_07	user_20	t	t	f
user_08	user_20	t	t	f
user_09	user_20	t	t	f
user_20	user_11	t	t	f
user_20	user_12	t	t	f
user_20	user_13	t	t	f

The Hierarchy



Membership Chaining

Membership can create a chain of privileges:

```
\c - user_11

CREATE TABLE drop_test4 (x INTEGER);

\c - user_07

-- user_07 is a member of user_20, and user_10 is a member of user_11
DROP TABLE drop_test4;

SET ROLE user_11;
```

7. Object Access Control

In most cases, when a role creates an object, only the owner role can view or modify the object; the objects include

- Databases
- Domains
- Foreign data wrappers
- Foreign servers
- Functions
- Languages
- Large objects
- Procedures
- Schemas
- Sequences
- Server parameters (added in PostgreSQL 15)
- Tables
- Tablespaces
- Types
- View

Object ownership can be modified with ALTER; for more details see https://www.postgresql.org/docs/current/ddl-priv.html.

Table Permissions

As an example, here are the GRANT operations that can be performed on tables:

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] table_name [, ...]
         | ALL TABLES IN SCHEMA schema_name [, ...] }
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { { SELECT | INSERT | UPDATE | REFERENCES } ( column_name [, ...] )
    [, ...] | ALL [ PRIVILEGES ] ( column_name [, ...] ) }
    ON [ TABLE ] table_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]
```

Granting Table Permissions

```
\c - user_01

CREATE TABLE grant_test (x INTEGER);

\dp
                            Access privileges
 Schema |    Name    | Type  | Access privileges | Column privileges | Policies
--------+------------+-------+-------------------+-------------------+----------
 public | drop_test3 | table | (null)            |                   |
 public | grant_test | table | (null)            |                   |

GRANT SELECT ON TABLE grant_test TO user_02;
```

Granting Table Permissions

```
\dp
                                Access privileges
 Schema |     Name     | Type  |     Access privileges     | Column privileges | Policies
--------+--------------+-------+---------------------------+-------------------+----------
 public | drop_test3   | table | (null)                    |                   |
 public | grant_test   | table | user_01=arwdDxt/user_01+| |                   |
        |              |       | user_02=r/user_01         |                   |
```

This blog entry explains the letters: https://momjian.us/main/blogs/pgblog/2019.html#February_4_2019.

Permission Letters

```
SELECT relacl FROM pg_class where relname = 'grant_test';
                   relacl
-------------------------------------------
 {user_01=arwdDxt/user_01,user_02=r/user_01}

SELECT (aclexplode(relacl)).grantee::regrole,
       (aclexplode(relacl)).privilege_type
FROM pg_class
WHERE relname = 'grant_test'
ORDER BY 1, 2;
 grantee | privilege_type
---------+----------------
 user_01 | DELETE
 user_01 | INSERT
 user_01 | REFERENCES
 user_01 | SELECT
 user_01 | TRIGGER
 user_01 | TRUNCATE
 user_01 | UPDATE
 user_02 | SELECT
```

Permission Aggregation

```
WITH acls AS (
    SELECT (aclexplode(relacl)).grantee::regrole AS rolename,
           (aclexplode(relacl)).privilege_type AS acl
    FROM pg_class
    WHERE relname = 'grant_test'
) SELECT rolename, array_agg(acl)
FROM acls
GROUP BY 1
ORDER BY 1, 2;
 rolename |                          array_agg
----------+-------------------------------------------------------------
 user_01  | {INSERT,SELECT,UPDATE,DELETE,TRUNCATE,REFERENCES,TRIGGER}
 user_02  | {SELECT}
```

This blog entry explains this aggregation: https://momjian.us/main/blogs/pgblog/2019.html#February_6_2019.

Using Information_schema

```
SELECT grantee,
       array_agg(privilege_type)
FROM  information_schema.role_table_grants
WHERE table_name = 'grant_test'
GROUP BY grantee
ORDER BY 1,2;
 grantee |                          array_agg
---------+-------------------------------------------------------------
 user_01 | {INSERT,SELECT,UPDATE,DELETE,TRUNCATE,REFERENCES,TRIGGER}
 user_02 | {SELECT}
```

The information_schema contains views for other object types.

Objects With Default Public Access

Most objects have permissions only for the object owner, and subsequent GRANT statements can be used to open permissions. However, the following objects allow public access by default:

- Connection and temporary table creation in databases
- Execution of functions and procedures
- Usage of languages
- Usage of data type and domains

When creating such objects, if public access is not desired, create the object and modify its access permissions in a single transaction block, as outlined at https://www.postgresql.org/docs/current/sql-createfunction.html.

8. Predefined Roles

As mentioned before, superuser roles are all-powerful, and sometimes such power is needed. However, often less powerful privileges are sufficient. To allow roles to be assigned some superuser permissions, but not others, predefined non-login roles have been created:

```
\du pg_*
                List of roles
          Role name          |  Attributes
-----------------------------+--------------
 pg_checkpoint               | Cannot login
 pg_create_subscription      | Cannot login
 pg_database_owner           | Cannot login
 pg_execute_server_program   | Cannot login
 pg_monitor                  | Cannot login
 pg_read_all_data            | Cannot login
 pg_read_all_settings        | Cannot login
 pg_read_all_stats           | Cannot login
 pg_read_server_files        | Cannot login
 pg_signal_backend           | Cannot login
 pg_stat_scan_tables         | Cannot login
 pg_use_reserved_connections | Cannot login
 pg_write_all_data           | Cannot login
 pg_write_server_files       | Cannot login
```

Predefined Role Membership

As mentioned before, superuser roles are all-powerful, and sometimes such power is needed. However, often less powerful privileges are sufficient. To allow roles to be assigned some superuser permissions, but not others, predefined non-login roles have been created:

```
\c - user_01

SELECT * FROM pg_authid;
ERROR:  permission denied for table pg_authid

\c - postgres
GRANT pg_read_all_data TO user_01;

\c - user_01
SELECT * FROM pg_authid;
  oid  |           rolname           | rolsuper …
-------+-----------------------------+----------…
    10 | postgres                    | t        …
  6171 | pg_database_owner           | f        …
  6181 | pg_read_all_data            | f        …
  6182 | pg_write_all_data           | f        …
```

Conclusion

https://momjian.us/presentations

https://www.flickr.com/photos/26424952@N00/