

# Access Control Overview



## Copyright

© Postgres Professional, 2017–2025

Authors: Egor Rogov, Pavel Luzanov, Ilya Bashtanov, Igor Gnatyuk

Translated by: Liudmila Mantrova, Alexander Meleshko, Elena Sharafutdinova

Photo: Oleg Bartunov (Phu Monastery and Bhrikuti Peak, Nepal)

## Use of Course Materials

Non-commercial use of course materials (presentations, demonstrations) is allowed without restrictions. Commercial use is possible only with the written permission of Postgres Professional. It is prohibited to make changes to the course materials.

## Feedback

Please send your feedback, comments and suggestions to:

[edu@postgrespro.ru](mailto:edu@postgrespro.ru)

## Disclaimer

Postgres Professional assumes no responsibility for any damages and losses, including loss of income, caused by direct or indirect, intentional or accidental use of course materials. Postgres Professional company specifically disclaims any warranties on course materials. Course materials are provided “as is,” and Postgres Professional company has no obligations to provide maintenance, support, updates, enhancements, or modifications.

Roles and Attributes  
Connecting to a Server  
Password Authentication  
Privileges and Privilege Management  
Role Categories  
Group and Predefined Roles  
Default Privileges  
Privileges and Routines

# Roles and Attributes

Role can be a DBMS user

not associated with the OS user

Roles can be included into other roles

simplifies access setup

Attributes define the properties of a role

LOGIN	can log in
SUPERUSER	superuser privileges
CREATEDB	can create databases
CREATEROLE	can create roles
and others	

Roles in PostgreSQL are used for two purposes. Firstly, a role can be a DBMS user. Secondly, roles can be members of other roles — it is convenient when setting up access.

Formally, roles are not associated with operating system users in any way, but many programs imply it when choosing default values. For example, if psql is started on behalf of the student OS user, the connection is established on behalf of the database role with the same name, i.e., student (unless another role is explicitly specified in the psql options).

At the time of cluster initialization, an initial role is defined, which has superuser privileges (this role is usually called postgres). Later on, you can create, modify, and delete roles.

<https://postgrespro.com/docs/postgresql/16/database-roles>

A role has several *attributes* which define its general properties and rights (unrelated to particular objects).

Generally, attributes come in two opposite variations; for example, CREATEDB (can create databases) and NOCREATEDB (not allowed to create databases).

A role with the LOGIN attribute is considered a user role. A role with NOLOGIN cannot connect to the server and is typically used for grouping other roles.

The table lists only some of the possible attributes.

<https://postgrespro.com/docs/postgresql/16/role-attributes>

<https://postgrespro.com/docs/postgresql/16/sql-createrole>

## Roles and Attributes

Create a role for the user Alice. Two attributes are specified.

Note which role the commands are executed as. The name of the current role is in the prompt.

```
student=# CREATE ROLE alice LOGIN PASSWORD 'alice';
```

```
CREATE ROLE
```

You can get the list of roles with the command:

```
student=# \du
```

List of roles	
Role name	Attributes
alice	
postgres	Superuser, Create role, Create DB, Replication, Bypass RLS
student	Superuser

Note that the student role is a superuser. This is why we did not have any access issues so far.

Create a database:

```
student=# CREATE DATABASE access_overview;
```

```
CREATE DATABASE
```

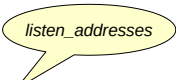
```
student=# \c access_overview
```

You are now connected to database "access\_overview" as user "student".

# Connecting to a Server

1. The lines of `pg_hba.conf` are searched from top to bottom
2. The first line that corresponds to the provided connection settings (type, database, user, address) will be used

#	TYPE	DATABASE	USER	ADDRESS	METHOD
	local	all	postgres		peer
	local	all	all		peer
	host	all	all	127.0.0.1/32	scram-sha-256
	host	all	all	::1/128	scram-sha-256
...					
	local — socket		all — any role		
	host — TCP/IP		role name		
...					
		all — any DB		all — any IP	
		database name		IP/mask	
				domain name	



5

For each new client, the server has to evaluate whether a database connection should be allowed.

Connection settings are defined in the `pg_hba.conf` configuration file (hba stands for host-based authentication). As with the main configuration file (`postgresql.conf`), changes come into effect only after the server reloads this file, either via the `pg_reload_conf()` SQL function or the `reload` command of the management utility.

When a new client appears, the server reads the configuration file from top to bottom to find the line that matches the requested connection. The match is defined by four fields: connection type, database name, user name, and IP address.

Here we list only the main basic options.

Connection: local (unix sockets) or host (a TCP/IP connection).

Database: all (corresponds to any database) or the name of a particular database.

User: all or the name of a particular role.

Address: all, IP address range, or a domain name. The address is omitted for the local connection type. By default, PostgreSQL listens only for connections coming from localhost; the `listen_addresses` parameter is usually set to `*` (listen on all interfaces), while the access is controlled using `pg_hba.conf` settings.

<https://postgrespro.com/docs/postgresql/16/client-authentication>

# Connecting to a Server

3. The server performs authentication using the chosen method
4. If successful, access is allowed; otherwise, it is forbidden (if no rows match the given parameters, access is forbidden)

#	TYPE	DATABASE	USER	ADDRESS	METHOD
	local	all	postgres		peer
	local	all	all		peer
	host	all	all	127.0.0.1/32	scram-sha-256
	host	all	all	:::1/128	scram-sha-256

trust — allow  
reject — forbid  
scram-sha-256 и md5 — request a password  
peer — ask OS

Once the server finds an appropriate line in the file, it performs client authentication using the method specified in this line, and checks for the LOGIN attribute and the CONNECT privilege. If everything is OK, the connection is allowed; otherwise, it is forbidden (other lines won't be considered in this case).

If no appropriate line is found, the access is also forbidden.

Thus, more specific connection lines should precede more generic ones.

There are a lot of different authentication methods:

<https://postgrespro.com/docs/postgresql/16/auth-methods>

Here are some of the main ones.

The trust method allows connections unconditionally. If security is not a concern, you can specify the trust method and use all for all the other parameters; then all connections will be allowed.

The reject method, on the contrary, unconditionally forbids connections.

The scram-sha-256 method asks for a password and checks that the provided password matches the one stored in the system catalog of the database cluster. The md5 method is considered deprecated.

The peer method checks the name of the operating system user and allows connections on behalf of the database user with the same name (you can also define a different name mapping).

## At the server side

- the password is set when the role is created and can be changed later
- a user that has no password won't be able to connect
- the password is stored in the pg\_authid system catalog

## Entering the password on the client

- manually
- using the PGPASSWORD environment variable
- using the ~/.pgpass file (lines format: *node:port:database:role:password*)

If password authentication is used, there must be a reference password stored for the user; otherwise the connection will be rejected.

Password hashes are stored in the pg\_authid table in the system catalog.

The user can either enter the password manually, or automate password input using one of the following options.

First, the password can be set in the PGPASSWORD environment variable (on the client). However, it is inconvenient if you have to connect to several databases, and it is not recommended for security reasons.

The second option to store passwords on the client is to use the ~/.pgpass file.

The access to this file must be allowed to its owner only (chmod 600), otherwise PostgreSQL will ignore it.

## Connection

In order for a role to connect to the database, it must have both the LOGIN attribute and the permission in the file `pg_hba.conf`. You can usually find it right beside the main configuration file:

```
student=# SHOW hba_file;

          hba_file 
-----
/etc/postgresql/16/main/pg_hba.conf
(1 row)
```

And you can read it directly from SQL:

```
student=# SELECT type, database, user_name, address, auth_method
FROM pg_hba_file_rules();
```

type	database	user_name	address	auth_method
local	{all}	{all}		trust
host	{all}	{all}	127.0.0.1	scram-sha-256
host	{all}	{all}	::1	scram-sha-256
local	{replication}	{all}		trust
host	{replication}	{all}	127.0.0.1	scram-sha-256
host	{replication}	{all}	::1	scram-sha-256

(6 rows)

(The contents may vary depending on the server build.)

We will use a TCP/IP (host) connection to localhost. Such connection corresponds to the second string in the output, expecting password-based authentication.

The role `alice` was created with a password right away, but the password can be changed at any time:

```
student=# ALTER ROLE alice PASSWORD 'alicepass';
```

ALTER ROLE

Attempt to connect using a connection string with all the info:

```
student$ psql 'host=localhost user=alice dbname=access_overview password=alicepass'
```

```
|  alice=> \conninfo
|
|  You are connected to database "access_overview" as user "alice" on host "localhost"
|  (address "127.0.0.1") at port "5432".
|  SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, compression: off)
```

Success!



Privileges define access rights of roles to objects

## Tables and views

SELECT	read data	} can be set per column
INSERT	insert rows	
UPDATE	change rows	
REFERENCES	foreign key (for tables)	
DELETE	delete rows	
TRUNCATE	empty (for tables)	
TRIGGER	create triggers	

Privileges establish a relation between subjects (roles) and objects in the cluster. They determine the actions that roles can perform with these objects.

There are different privileges available for different object types. This slide and the following one list privileges for basic database objects.

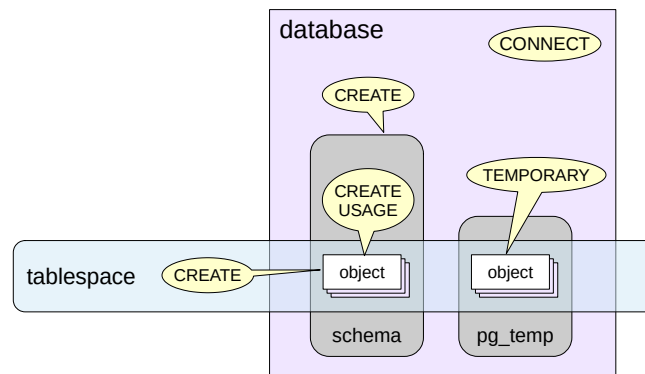
The widest choice of privileges is available for tables and views. Some of these privileges can be set not only at the table level, but also at the column level.

<https://postgrespro.com/docs/postgresql/16/ddl-priv>

<https://postgrespro.com/docs/postgresql/16/sql-grant>

# Privileges

Tablespaces,  
database, schemas



Sequences

SELECT	currval		
UPDATE		nextval	setval
USAGE	currval	nextval	

Sequences have a somewhat unexpected set of privileges. They serve to allow or restrict access to the three control functions.

For tablespaces, there is a CREATE privilege that allows the creation of objects in this tablespace.

For databases, the CREATE privilege allows you to create schemas in this database, and for a schema, the CREATE privilege allows you to create objects in this schema.

Since the exact name of the schema for temporary objects is unknown in advance, the privilege to create temporary tables has been moved to the database level (TEMPORARY).

The USAGE schema privilege allows access to objects in this schema.

The CONNECT database privilege allows connection to this database.

## Superusers

full access to all objects, no checks performed

## Owners

initially, all privileges on the object (can be revoked)

actions that are not regulated by privileges, such as deleting objects, granting and revoking privileges, etc.

## Other roles

access within the granted privileges

Generally speaking, a role's ability to access an object is defined by the role's privileges. But it makes sense to single out three categories of roles and discuss them separately.

1. Roles with the SUPERUSER attribute (superusers). These roles can do anything and ignore all access control checks.
2. Object owner. Initially, this is the role that created the object, although it can be changed later. It's not just the object creator role that becomes the owner, but also any other role included in it. The object owner gets the full range of privileges on this object.

Technically, these privileges can be revoked, but the owner always retains inherent rights on the actions that are not regulated by any privileges. In particular, the owner can grant and revoke privileges (including to and from themselves), delete the object, etc.

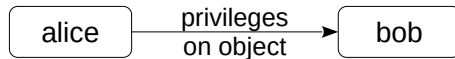
3. All other roles have access to the object only as far as the privileges granted to them allow it.

To check if a role has the necessary privilege with respect to some object, you can use the `has_*_privilege` functions:

<https://postgrespro.com/docs/postgresql/16/functions-info>

## Granting privileges

```
alice: GRANT privileges ON object TO bob;
```



## Revoking privileges

```
alice: REVOKE privileges ON object FROM bob;
```

The right to grant and revoke privileges on an object belongs to the owner of that object (and the superuser).

The syntax of the GRANT and REVOKE commands is quite complex. You can specify both individual and all possible privileges, both individual objects and groups of objects included in certain schemas, etc.

<https://postgrespro.com/docs/postgresql/16/sql-grant>

<https://postgrespro.com/docs/postgresql/16/sql-revoke>

## Privileges

Alice has connected to the database. Now she wants to create a schema and some objects.

```
|  alice=> CREATE SCHEMA alice;
|  ERROR:  permission denied for database access_overview
```

What's the issue?

---

Alice does not have the privilege to create schemas in the database. Grant it:

```
student=# GRANT CREATE ON DATABASE access_overview TO alice;
```

GRANT

Try again:

```
|  alice=> CREATE SCHEMA alice;
|  CREATE SCHEMA
```

Now, since Alice is the owner of the schema, she has all the privileges on it and can create any objects in it. This schema will be used by default:

```
|  alice=> SELECT current_schemas(false);
|
|  current_schemas
|  -----
|  {alice,public}
|  (1 row)
```

Alice creates two tables.

```
|  alice=> CREATE TABLE t1(n numeric);
|  CREATE TABLE
|  alice=> INSERT INTO t1 VALUES (1);
|  INSERT 0 1
|  alice=> CREATE TABLE t2(n numeric, who text DEFAULT current_user);
|  CREATE TABLE
|  alice=> INSERT INTO t2(n) VALUES (1);
|  INSERT 0 1
```

The superuser creates another role for the user Bob, who will access objects belonging to Alice.

```
student=# CREATE ROLE bob LOGIN PASSWORD 'bobpass';
```

CREATE ROLE

```
student$ psql 'host=localhost user=bob dbname=access_overview password=bobpass'
```

Bob attempts to access the table t1:

```
||  bob=> SELECT * FROM alice.t1;
||
||  ERROR:  permission denied for schema alice
||  LINE 1: SELECT * FROM alice.t1;
||                      ^
```

What happened?

---

Bob is neither a superuser nor the owner of this schema, so access is denied.

This command lists current access rights (Access privileges column):

```
|  alice=> \x \dn+ \x
```

```
Expanded display is on.
List of schemas
-[ RECORD 1 ]-----+-----
Name           | alice
Owner           | alice
Access privileges |
Description    |
-[ RECORD 2 ]-----+-----
Name           | public
Owner           | pg_database_owner
Access privileges | pg_database_owner=UC/pg_database_owner+
                | =U/pg_database_owner
Description    | standard public schema

Expanded display is off.
```

Privileges are displayed in the format: role=privileges/granted\_by.

Each privilege is encoded with a single character. In particular, for schemas:

- U = usage;
- C = create.

If the role name is omitted (as in the last line), the pseudorole public is implied. Note that the public pseudorole has only USAGE privilege on the public schema. Here, pg\_database\_owner is the owner of the database.

If the entire field is omitted (as in the first line), then the default privileges are implied: Alice gets both available privileges on her schema, and the other roles do not get any.

Grant Bob access to the schema. Alice can do that as the owner.

```
alice=> GRANT CREATE, USAGE ON SCHEMA alice TO bob;

GRANT
```

Bob attempts to access the table t1 again:

```
bob=> SELECT * FROM alice.t1;

ERROR:  permission denied for table t1
```

Another error. What happened now?

This time, Bob has access to the schema, but not to the table itself. This command shows who has access to the table:

```
alice=> \dp alice.t1
```

Access privileges					
Schema	Name	Type	Access privileges	Column privileges	Policies
alice	t1	table			

(1 row)

The field is empty: only the owner, Alice, has access.

Alice grants Bob read and update access:

```
alice=> GRANT SELECT, UPDATE ON alice.t1 TO bob;

GRANT
```

And read and insert rights for one column in the second table:

```
alice=> GRANT SELECT(n), INSERT ON alice.t2 TO bob;

GRANT
```

The privileges have changed:

```
alice=> \dp alice.*
```

Access privileges					
Schema	Name	Type	Access privileges	Column privileges	Policies
alice	t1	table	alice=arwdDxt/alice+ bob=rw/alice		
alice	t2	table	alice=arwdDxt/alice+ bob=a/alice	n: bob=r/alice	+

(2 rows)

Now the empty field has “appeared” and contains a complete list of privileges. Below are the designations, not all of them are quite

obvious:

- a = insert
- r = select
- w = update
- d = delete
- D = truncate
- x = reference
- t = trigger

Privileges for columns are displayed separately (under Column privileges).

---

This time Bob is successful. He adds the schema name to his search path, so that he does not have to type it in every time.

```
|| bob=> ALTER ROLE bob SET search_path = public, alice;
|| ALTER ROLE
```

Now, the search path will be set in each of Bob's sessions.

```
|| bob=> \c
|| You are now connected to database "access_overview" as user "bob".
|| bob=> SHOW search_path;
||
|| search_path
|| -----
|| public, alice
|| (1 row)
||
|| bob=> UPDATE t1 SET n = n + 1;
|| UPDATE 1
|| bob=> SELECT * FROM t1;
||
|| n
|| ---
|| 2
|| (1 row)
```

However, all other operations remain restricted.

```
|| bob=> DELETE FROM t1;
|| ERROR: permission denied for table t1
```

And the second table:

```
|| bob=> INSERT INTO t2(n) VALUES (100);
|| INSERT 0 1
|| bob=> SELECT n FROM t2;
||
|| n
|| ----
|| 1
|| 100
|| (2 rows)
```

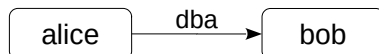
Reading of the other column is not permitted:

```
|| bob=> SELECT * FROM t2;
|| ERROR: permission denied for table t2
```

# Granting Membership

## Granting membership

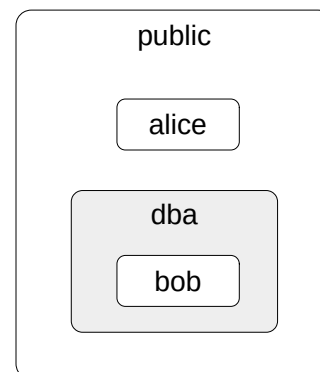
```
alice: GRANT dba TO bob;
```



public pseudo-role implicitly includes all other roles

## Revoking membership

```
alice: REVOKE dba FROM bob;
```



14

Any role can include other roles as members. In this case, the role acts as a group. PostgreSQL does not have a separate “group” entity.

A role can be a member of multiple roles; in turn, a member role may include other roles, but circular dependencies are not permitted.

By default, a role inherits the privileges of any group roles it is a member of. This behavior can be changed by using the NOINHERIT attribute for the role. This makes the user have to explicitly switch to the group role using SET ROLE in order to use its privileges. Role attributes are not inherited, but it is possible to switch to a parent role to use its attributes.

Roles that include other roles typically have the NOLOGIN attribute and are referred to as “group roles”. We can think of a group role as a predefined set of privileges that can be granted to a role just like any regular privilege. This simplifies access control and administration.

There is also a pseudorole called public, which implicitly includes all the other roles. Any privilege granted to the public role is automatically granted to all the other roles as well.

<https://postgrespro.com/docs/postgresql/16/role-membership>



# Predefined Roles

<code>pg_read_all_settings</code>	read all server parameters	} <code>pg_monitor</code>
<code>pg_read_all_stats</code>	access statistics	
<code>pg_stat_scan_tables</code>	monitoring and locking tables	
<code>pg_read_all_data</code>	read data in all tables	
<code>pg_write_all_data</code>	change data in all tables	
<code>pg_read_server_files</code>	read files on the server	
<code>pg_write_server_files</code>	write to files on the server	
<code>pg_execute_server_programs</code>	run programs on the server	
...		

PostgreSQL provides a number of predefined roles that have access to some frequently used, but privileged capabilities and data. Membership in these roles may be granted to users in order to facilitate administrative and maintenance tasks without providing them superuser capabilities.

The list of predefined roles increases with each PostgreSQL version. The full list of all the roles, including predefined ones, can be seen by `\duS` command in `psql`.

<https://postgrespro.com/docs/postgresql/16/predefined-roles>

You can create your own administrative group roles, e.g. for managing backups.

## Group Roles

Bob could not read the second column:

```
|| bob=> SELECT * FROM t2;
|| ERROR: permission denied for table t2
```

The superuser includes Bob into the predefined group role pg\_read\_all\_data:

```
student=# GRANT pg_read_all_data TO bob;
```

GRANT ROLE

Now Bob can read all tables as if he had been granted SELECT privileges for all tables and USAGE privileges for all schemas:

```
|| bob=> SELECT * FROM t2;
||
||      n | who
|| -----+-----
||      1 | alice
||     100 | bob
|| (2 rows)
```

You can use the command \drg in psql to get role membership information:

```
student=# \drg
```

```

              List of role grants
Role name | Member of | Options | Grantor
-----+-----+-----+-----
bob       | pg_read_all_data | INHERIT, SET | postgres
(1 row)
```

The Options column shows attributes for the membership. For Bob as a member of pg\_read\_all\_data, SET means the right to switch to the group role, and INHERIT means that Bob can use the group privileges without explicitly switching to the group.

Exclude Bob from the group:

```
student=# REVOKE pg_read_all_data FROM bob;
```

REVOKE ROLE

The only privilege for functions and procedures

EXECUTE	execution
---------	-----------

Security modes

SECURITY INVOKER	executed with the privileges of the caller (by default)
SECURITY DEFINER	executed with the privileges of the owner

The privilege EXECUTE, the only privilege for functions and procedures, allows them to execute.

The user on behalf of which the routine is executed is important. If a routine is declared as a SECURITY INVOKER (by default), it is executed with the rights of the user that runs it. In this case, the statements inside the routine can access only those objects that are accessible to the calling user.

On the other hand, if declared with the SECURITY DEFINER, the routine will use the rights of its owner. This is a way to allow certain users perform certain actions on objects they personally have no access to.

<https://postgrespro.com/docs/postgresql/16/sql-createfunction>

<https://postgrespro.com/docs/postgresql/16/sql-createprocedure>

## Privileges of the public pseudo-role

- connect to any database
- access to the system catalog
- execution of any routines
- privileges are granted automatically on each new object

## Configurable default privileges

- the possibility to additionally grant or revoke privileges on a newly created object

18

As we have already said, the public pseudo-role includes all other roles, so they inherit all the privileges granted to public.

And public has quite an extensive list of privileges by default. In particular:

- The right to connect to any database (that's why the role alice could connect to the database although the CONNECT privilege had not been explicitly granted to this role)
- Access to the system catalog
- Execution of any routines

On the one hand, it enables seamless operation without having to deal with privileges; but on the other hand, it brings extra complications if access control is really required.

The public role automatically receives all the privileges listed above for all newly created objects. So it is not enough to simply revoke the EXECUTE privilege from public: once a new routine appears, public immediately gets the right to execute it.

There is a special mechanism of default privileges that enables you to automatically grant and revoke the required privileges on newly created objects. It can be also used to revoke the EXECUTE privilege from the public pseudo-role.

<https://postgrespro.com/docs/postgresql/16/sql-alterdefaultprivileges>

## Configurable Default Privileges

Alice creates a function:

```
alice=> CREATE FUNCTION foo() RETURNS SETOF t2
AS $$
SELECT * FROM t2;
$$ LANGUAGE sql STABLE;

CREATE FUNCTION
```

Can Bob execute the function without Alice granting him the EXECUTE privilege?

```
bob=> SELECT foo();

ERROR: permission denied for table t2
CONTEXT: SQL function "foo" statement 1
```

Bob can call the function, but still cannot access any objects for which he does not have the appropriate privileges.

If Bob creates a table t2 in the schema public, the function will work for both users but will actually access different tables, since Alice and Bob have different search paths.

In order for Bob to create a table in the schema public, he must have the CREATE privilege for the schema (in PostgreSQL 15+):

```
student=# GRANT CREATE ON SCHEMA public TO bob;
```

GRANT

```
bob=> CREATE TABLE t2(n numeric, who text DEFAULT current_user);

CREATE TABLE

bob=> INSERT INTO t2(n) VALUES (42);

INSERT 0 1

bob=> SELECT foo();

      foo
-----
(42,bob)
(1 row)

alice=> SELECT foo();

      foo
-----
(1,alice)
(100,bob)
(2 rows)
```

Alternatively, the function can be declared as using the privileges of its owner:

```
alice=> ALTER FUNCTION foo() SECURITY DEFINER;

ALTER FUNCTION
```

In this case, the function will always be executed with the privileges Alice has, regardless of who actually calls it.

Bob deletes his table...

```
bob=> DROP TABLE t2;

DROP TABLE
```

...and gets access to the Alice's table:

```
bob=> SELECT foo();

      foo
-----
(1,alice)
(100,bob)
(2 rows)
```

In this situation, Alice needs to keep an eye on the granted privileges. For one, she should revoke the EXECUTE privilege from the role public and explicitly grant it only to the roles that should have it.

```

|  alice=> REVOKE EXECUTE ON ALL ROUTINES IN SCHEMA alice FROM public;
|
|  REVOKE
|
||  bob=> SELECT foo();
||
||  ERROR:  permission denied for function foo

```

To make matters worse, for each new function, the EXECUTE privilege is always granted to public by default.

-----

You can configure default privileges so that specific users would get (or lose) specific privileges on newly created objects:

```

|  alice=> ALTER DEFAULT PRIVILEGES
|  FOR ROLE alice
|  REVOKE EXECUTE ON ROUTINES FROM public;
|
|  ALTER DEFAULT PRIVILEGES
|
|  alice=> ALTER DEFAULT PRIVILEGES
|  FOR ROLE alice
|  GRANT EXECUTE ON ROUTINES TO bob;
|
|  ALTER DEFAULT PRIVILEGES
|
|  alice=> \ddp
|
|          Default access privileges
|  Owner | Schema |   Type   | Access privileges
|-----+-----+-----+-----+
|  alice |        | function | alice=X/alice    +
|        |        |          | bob=X/alice
| (1 row)

```

Now, Bob is immediately granted execute privileges on routines created by Alice, while other users will not have permission to run them.

```

|  alice=> CREATE FUNCTION bar() RETURNS integer
|  LANGUAGE sql IMMUTABLE SECURITY DEFINER
|  RETURN 1;
|
|  CREATE FUNCTION
|
||  bob=> SELECT bar();
||
||  bar
||  ----
||  1
|| (1 row)

```

Roles, attributes and privileges together form a flexible mechanism that allows you to set up access control in different ways

- can grant all access widely
- can restrict access heavily if necessary

When creating a new role, you need to ensure that it can connect to the server

1. Create two roles (the password must match the name):
  - employee — store employee,
  - buyer — customer.Make sure that the created roles can connect to the database.
2. Revoke the rights of the role public to perform all functions and connect to the database.
3. Delimit access in such a way that:
  - the employee could only order books and add authors and books,
  - the buyer could only purchase books.Check the changes in the application.

1. The employee is an internal user of the application, authentication is performed at the database level.

The buyer is an external user. In a real online store, the management of such users falls to the application, and all queries come to the database from one “generalized” role (buyer). The identifier of a specific customer can be passed as a parameter (but we do not do this in our application).

3. Generally speaking, access control should be embedded in the application. In our educational application, it is not implemented on purpose: instead, on the web page, you can explicitly select the role on behalf of which the query will be sent to the database.

This allows you to see how the server side will behave if the application is not working correctly.

So, users need to get:

- The right to connect to the bookstore database and access the bookstore schema.
- Access to views that are accessed directly.
- Access to functions that are called as part of the API. If you leave the SECURITY INVOKER functions in, you will have to grant access to all the “underlying” objects (tables, other functions). However, it is more convenient to simply declare API functions as SECURITY DEFINER.

Of course, roles need to be granted privileges only on those objects that they should have access to.



## 1. Create roles

```
=> CREATE ROLE employee LOGIN PASSWORD 'employee';
```

CREATE ROLE

```
=> CREATE ROLE buyer LOGIN PASSWORD 'buyer';
```

CREATE ROLE

The default settings allow local connections by password. This works for us.

## 2. Privileges of “public”

Revoke extra privileges from the role “public”.

```
=> REVOKE EXECUTE ON ALL FUNCTIONS IN SCHEMA bookstore FROM public;
```

REVOKE

```
=> REVOKE CONNECT ON DATABASE bookstore FROM public;
```

REVOKE

## 3. Access control

Functions executed with the creator’s privileges.

```
=> ALTER FUNCTION get_catalog(text,text,boolean) SECURITY DEFINER;
```

ALTER FUNCTION

```
=> ALTER FUNCTION update_catalog() SECURITY DEFINER;
```

ALTER FUNCTION

```
=> ALTER FUNCTION add_author(text,text,text) SECURITY DEFINER;
```

ALTER FUNCTION

```
=> ALTER FUNCTION add_book(text,integer[]) SECURITY DEFINER;
```

ALTER FUNCTION

```
=> ALTER FUNCTION buy_book(integer) SECURITY DEFINER;
```

ALTER FUNCTION

```
=> ALTER FUNCTION book_name(integer,text,integer) SECURITY DEFINER;
```

ALTER FUNCTION

```
=> ALTER FUNCTION authors(books) SECURITY DEFINER;
```

ALTER FUNCTION

Buyer privileges: the buyer can search for and purchase books.

```
=> GRANT CONNECT ON DATABASE bookstore TO buyer;
```

GRANT

```
=> GRANT USAGE ON SCHEMA bookstore TO buyer;
```

GRANT

```
=> GRANT EXECUTE ON FUNCTION get_catalog(text,text,boolean) TO buyer;
```

GRANT

```
=> GRANT EXECUTE ON FUNCTION buy_book(integer) TO buyer;
```

GRANT

Employee privileges: the employee must have access to view and add books and authors, as well as to the catalog for ordering more books.

```
=> GRANT CONNECT ON DATABASE bookstore TO employee;
```

GRANT

```
=> GRANT USAGE ON SCHEMA bookstore TO employee;

GRANT

=> GRANT SELECT,UPDATE(onhand_qty) ON catalog_v TO employee;

GRANT

=> GRANT SELECT ON authors_v TO employee;

GRANT

=> GRANT SELECT ON operations_v TO employee;

GRANT

=> GRANT EXECUTE ON FUNCTION book_name(integer,text,integer) TO employee;

GRANT

=> GRANT EXECUTE ON FUNCTION authors(books) TO employee;

GRANT

=> GRANT EXECUTE ON FUNCTION author_name(text,text,text) TO employee;

GRANT

=> GRANT EXECUTE ON FUNCTION add_book(text,integer[]) TO employee;

GRANT

=> GRANT EXECUTE ON FUNCTION add_author(text,text,text) TO employee;

GRANT
```

1. Register the user roles alice and bob.
2. Modify the `pg_hba.conf` file to allow access without password only for the postgres and student users, ensuring that access for alice and bob remains restricted.
3. Enable peer authentication method for alice and bob. Check that connection attempts fail without OS user mapping. Create such mapping for alice.
4. Check the capability to use the same OS user mapping for different database roles.

2. Using a text editor, insert the new entry before the first uncommented directive in `pg_hba.conf`.

```
local all postgres,student trust
```

Reload configuration.

3. Modify the inserted directive in `pg_hba.conf` by replacing it with the following:

```
local all postgres,student trust
local all alice,bob peer
```

Append the following to the end of the `pg_ident.conf` file:

```
stmap student alice
```

4. For alice and bob roles to share a single mapping, the `pg_ident.conf` file shall be as follows:

```
stmap student alice
stmap student bob
```

## 1. Add Roles

Register roles with the permission to log in.

```
=> CREATE ROLE alice LOGIN;
```

CREATE ROLE

```
=> CREATE ROLE bob LOGIN;
```

CREATE ROLE

## 2. Limit the use of trust

Edit the contents of pg\_hba.conf, allowing the trust method only for postgres and student.

```
student$ sudo sed -i 's/^local.*all.*all.*trust.*$/local all postgres,student trust\n/'  
/etc/postgresql/16/main/pg_hba.conf
```

This is what we get:

```
=> SELECT type,database,user_name,address,auth_method,error  
FROM pg_hba_file_rules  
ORDER BY rule_number;
```

type	database	user_name	address	auth_method	error
local	{all}	{postgres,student}		trust	
host	{all}	{all}	127.0.0.1	scram-sha-256	
host	{all}	{all}	::1	scram-sha-256	
local	{replication}	{all}		trust	
host	{replication}	{all}	127.0.0.1	scram-sha-256	
host	{replication}	{all}	::1	scram-sha-256	

(6 rows)

Reload the configuration.

```
=> SELECT pg_reload_conf();
```

```
pg_reload_conf  
-----  
t  
(1 row)
```

Now, neither alice nor bob can connect.

```
student$ psql -l -U alice
```

```
psql: error: connection to server on socket "/var/run/postgresql/.s.PGSQL.5432" failed:  
FATAL: no pg_hba.conf entry for host "[local]", user "alice", database "postgres", no  
encryption
```

```
student$ psql -l -U bob
```

```
psql: error: connection to server on socket "/var/run/postgresql/.s.PGSQL.5432" failed:  
FATAL: no pg_hba.conf entry for host "[local]", user "bob", database "postgres", no  
encryption
```

## 3. Peer Authentication Method

Using a text editor, add another line with the peer authentication method to allow alice and bob to connect.

```
student$ sudo sed -i '/^local.*all.*postgres,student.*$/a local all alice,bob peer'  
/etc/postgresql/16/main/pg_hba.conf
```

Contents of pg\_hba.conf:

```
=> SELECT type,database,user_name,address,auth_method,error  
FROM pg_hba_file_rules  
ORDER BY rule_number;
```

type	database	user_name	address	auth_method	error
local	{all}	{postgres,student}		trust	
local	{all}	{alice,bob}		peer	
host	{all}	{all}	127.0.0.1	scram-sha-256	
host	{all}	{all}	::1	scram-sha-256	
local	{replication}	{all}		trust	
host	{replication}	{all}	127.0.0.1	scram-sha-256	
host	{replication}	{all}	::1	scram-sha-256	

(7 rows)

Reload the configuration.

```
=> SELECT pg_reload_conf();
```

```
pg_reload_conf
-----
t
(1 row)
```

The users still cannot log in, but the error message is now different.

```
student$ psql -l -U alice
```

```
psql: error: connection to server on socket "/var/run/postgresql/.s.PGSQL.5432" failed:
FATAL: Peer authentication failed for user "alice"
```

```
student$ psql -l -U bob
```

```
psql: error: connection to server on socket "/var/run/postgresql/.s.PGSQL.5432" failed:
FATAL: Peer authentication failed for user "bob"
```

The peer authentication method requires the user names in the OS and roles in PostgreSQL to match. Map the role alice to the OS user student by adding a line to pg\_ident.conf. Do not set up the mapping for bob, though.

```
student$ echo 'stmap student alice' | sudo tee -a /etc/postgresql/16/main/pg_ident.conf
```

```
stmap student alice
```

Add the parameter map to the line to set the mapping.

```
student$ sudo sed -i 's/peer.*/peer map=stmap/' /etc/postgresql/16/main/pg_hba.conf
```

Contents of pg\_hba.conf:

```
=> SELECT type,database,user_name,address,auth_method,options,error
FROM pg_hba_file_rules
ORDER BY rule_number;
```

type	database	user_name	address	auth_method	options	error
local	{all}	{postgres,student}		trust		
local	{all}	{alice,bob}		peer	{map=stmap}	
host	{all}	{all}	127.0.0.1	scram-sha-256		
host	{all}	{all}	::1	scram-sha-256		
local	{replication}	{all}		trust		
host	{replication}	{all}	127.0.0.1	scram-sha-256		
host	{replication}	{all}	::1	scram-sha-256		

(7 rows)

Reload the configuration.

```
=> SELECT pg_reload_conf();
```

```
pg_reload_conf
-----
t
(1 row)
```

Now alice can connect to the database and execute commands.

```
student$ psql -c '\conninfo' -U alice -d student
```

```
You are connected to database "student" as user "alice" via socket in
"/var/run/postgresql" at port "5432".
```

And bob cannot.

```
student$ psql -c '\conninfo' -U bob -d student
```

```
psql: error: connection to server on socket "/var/run/postgresql/.s.PGSQL.5432" failed:
FATAL: Peer authentication failed for user "bob"
```

#### 4. One Mapping for Multiple Roles

Allow bob to connect under the same conditions as alice.

```
student$ echo 'stmap student bob' | sudo tee -a /etc/postgresql/16/main/pg_ident.conf
```

```
stmap student bob
```

Lines added to pg\_ident.conf:

```
student$ sudo tail -n2 /etc/postgresql/16/main/pg_ident.conf
```

```
stmap student alice
stmap student bob
```

Reload the configuration.

```
=> SELECT pg_reload_conf();
```

```
pg_reload_conf
-----
t
(1 row)
```

Now both alice and bob can connect.

```
student$ psql -c '\conninfo' -U alice -d student
```

```
You are connected to database "student" as user "alice" via socket in
"/var/run/postgresql" at port "5432".
```

```
student$ psql -c '\conninfo' -U bob -d student
```

```
You are connected to database "student" as user "bob" via socket in "/var/run/postgresql"
at port "5432".
```