

# Access Control

## Access Control Overview



15

### Copyright

© Postgres Professional, 2023

Authors: Egor Rogov, Pavel Luzanov, Ilya Bashtanov

Translated by Liudmila Mantrova

Cover photo by 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.

# Agenda



Roles and attributes  
Connecting to a server  
Privileges  
Row-level security policies

# Roles and Attributes



A role is a database user

roles are not associated with OS users

Role properties are defined by attributes

LOGIN	permission to connect
PASSWORD	a password
SUPERUSER	permission for everything
CREATEDB	permission to create databases
CREATEROLE	permission to create roles
REPLICATION	permission to use the replication protocol
etc.	

3

A role is a database user. (A role can also comprise a *group* of users, but we'll discuss it later.)

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/15/database-roles>

A role has several *attributes* that define its general properties and rights (unrelated to object access).

There are usually two flavors of each attribute; for example, CREATEDB (gives the right to create a database) and NOCREATEDB (gives no such right). As a rule, a restrictive flavor is the default one.

If a role has no LOGIN attribute, it cannot connect to a server. (Such roles can be used as group ones.)

This slide lists only some of the available attributes. INHERIT and BYPASSRLS attributes will be covered in more detail further in this lecture.

<https://postgrespro.com/docs/postgresql/15/role-attributes>

<https://postgrespro.com/docs/postgresql/15/sql-createrole>

## Roles and Attributes

Let's create a role for user Alice. The command specifies two attributes.

In the context of this demo, it is important to see the name of the role that executes commands, so the name of the current role is displayed in the prompt.

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

```
CREATE ROLE
```

The following command displays the list of roles:

```
student=# \du
```

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

Note that the student role is a superuser. That's why there has been no need to take care of access rights so far.

Let's create a database as well:

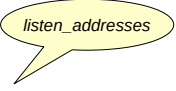
```
student=# CREATE DATABASE access_overview;
```

```
CREATE DATABASE
```

# Connection

1. The rows of `pg_hba.conf` are searched from top to bottom
2. The first row that corresponds to the provided connection parameters (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 database		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 parameters 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 (`SELECT pg_reload_conf()` in SQL, or `pg_ctl reload` in the operating system terminal).

When a new client appears, the server reads the configuration file from top to bottom to find the row 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, unavailable for Windows) or `host` (a TCP/IP connection).

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

User: `all` or the name of a particular role.

Address: `all`, a particular IP address with a subnet mask, or a domain name. The address is omitted for the `local` connection type. By default, PostgreSQL listens for incoming connections only on 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/15/client-authentication>

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 all  
reject — forbid all  
scram-sha-256 and md5 — request a password  
peer — ask OS

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

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

Thus, more specific connection rows should precede more generic ones while the file is viewed from top to bottom.

There are a lot of different authentication methods:

<https://postgrespro.com/docs/postgresql/15/auth-methods>

Here we mention only 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 all 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 pattern).

## At the server side

- the password is set when the role is created and can be altered later
- a user that has no password won't be able to connect
- the password is stored in the `pg_authid` table of the system catalog

## Entering the password on the client

- manually
- using the `PGPASSWORD` environment variable
- using the `~/.pgpass` file (its lines have the following 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.

Passwords are stored in the `pg_authid` table of the system catalog.

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

The first one is to define the password on the client in the `PGPASSWORD` environment variable. 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, otherwise PostgreSQL will ignore it.

## Connection

To be able to connect to a database, a role must have the LOGIN attribute, and the pg\_hba.conf file must allow connections for this role. The pg\_hba.conf file can be read 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}	{postgres}		trust
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

(7 rows)

(Depending on the PostgreSQL distribution, the contents of the file may differ.)

We are going to use a TCP/IP host connection. This connection type corresponds to the third line in pg\_hba.conf. It requires password authentication.

The alice role was created with a password, but you can change it any time:

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

ALTER ROLE

Let's try to connect to the database by providing all the required information in the connection string:

```
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 roles' access rights for different objects

## Tables and views

SELECT	read data	} can be used for columns
INSERT	insert rows	
UPDATE	update rows	
REFERENCES	set a foreign key	
DELETE	delete rows	
TRUNCATE	truncate a table	
TRIGGER	create triggers	

Privileges are defined for combinations of roles and database objects. They determine the actions that roles can perform with these objects.

There are different privileges available for different object types.

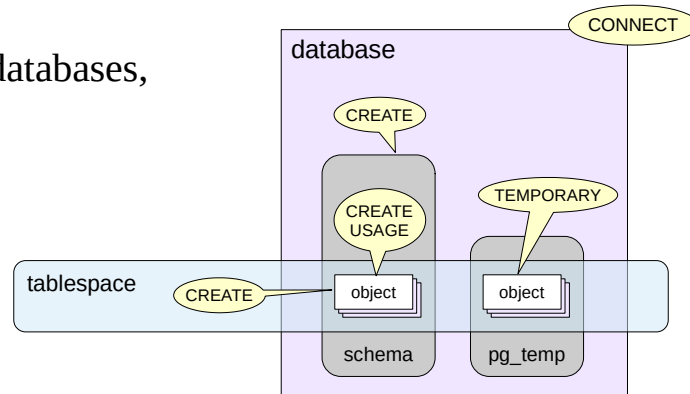
The widest choice of privileges is available for tables and views. Some of these privileges can be defined not only at the table level, but also at the column level.

<https://postgrespro.com/docs/postgresql/15/ddl-priv>

<https://postgrespro.com/docs/postgresql/15/sql-grant>

# Privileges

Tablespaces, databases,  
and schemas



Functions and procedures

EXECUTE

etc.

For tablespaces, there is a CREATE privilege that allows creating objects in this tablespace.

When defined for a database, the CREATE privilege allows creating schemas in this database; for schemas, this privilege allows creating objects in this schema.

Since the exact name of the schema for temporary objects is unknown in advance, the privilege for creating temporary tables is defined at the database level (TEMPORARY).

The USAGE privilege of a schema enables access to objects in this schema.

The CONNECT privilege of a database allows connections to this database.

## Superusers

unlimited access to all objects: no checks are performed

## Object owners

initially, all the privileges that can be granted on this object  
actions on the owned object that are not regulated by privileges,  
such as deleting objects or granting and revoking privileges

## Other roles

access rights are defined by the granted privileges

In general, we can simply say that the role's access to an object is defined by its privileges. But it makes sense to single out three categories of roles and discuss them separately.

1. Roles with the superuser attribute follow the most straightforward rules. Such roles bypass access control checks and can perform any operations.

2. Each object has an owner, i.e., the role that owns this object. Initially, it is the role that created the object, but you can change the owner later.

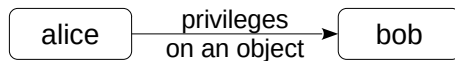
The object owner gets the full range of privileges on this object.

In theory, these privileges can be revoked, but the object owner also has an inalienable right to perform some actions that are not regulated by privileges. In particular, the owner can grant and revoke privileges (to other roles and to itself) and delete the owned object.

3. All the other roles can access objects as defined by the privileges granted to them.

## Granting privileges

```
alice=> GRANT privileges ON objects TO bob;
```



## Revoking privileges

```
alice=> REVOKE privileges ON object FROM bob;
```

Privileges on a particular object can be granted and revoked by the object owner (and a superuser).

The syntax of GRANT and REVOKE commands is quite complex. It allows addressing different scenarios: you can specify either some particular privileges or all the available privileges at once, grant privileges on individual objects or on groups of objects that belong to particular schemas, etc.

<https://postgrespro.com/docs/postgresql/15/sql-grant>

<https://postgrespro.com/docs/postgresql/15/sql-revoke>

## Privileges

Alice has managed to connect to the database. Now she wants to create a separate schema and several objects in it.

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

What has gone wrong?

---

Alice has no privilege to create schemas in this database. Let's grant it:

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

Now try once again:

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

Since Alice owns a separate schema, this role now has all the privileges for this schema and can create any objects in it. It is this schema that will be used by default:

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

Let's create 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
```

Now let's create another role for Bob, who is going to access the objects that belong to Alice.

```
student=# CREATE ROLE bob LOGIN PASSWORD 'bobpass';
CREATE ROLE
student$ psql "host=localhost user=bob dbname=access_overview password=bobpass"
```

Bob tries to access the t1 table.

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

What has caused an error?

---

The schema is unavailable because Bob is neither its owner nor a superuser.

To check access rights for a schema, you can run the following command (see the Access privileges column):

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

List of schemas			
Name	Owner	Access privileges	Description
alice	alice		
public	pg_database_owner	pg_database_owner=UC/pg_database_owner+=U/pg_database_owner	standard public schema
(2 rows)			

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

Each privilege is represented by one symbol; for example, schemas use the following notation:

- U = usage
- C = create

If the role name is omitted (as in the last row), the public pseudorole is implied.

If the whole field is omitted (as in the first row), the default privileges of the owner are in effect. We can see that alice has both privileges for her own schema.

Let's grant Bob access to the schema. Alice can do it as the schema owner.

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

Bob tries to access the table again:

```
bob=> SELECT * FROM alice.t1;
ERROR: permission denied for table t1
```

What has caused an error?

Now Bob has access to the schema, but has no access to the table itself. Here is how we can check access rights:

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

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

(1 row)

We see an empty field again: only the owner (i.e., Alice) has access permissions.

Alice grants Bob the right to read and update table t1:

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

For the second table, Bob gets the right to insert rows and read one of the columns:

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

Let's see how 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+	n:	+
			bob=a/alice	bob=r/alice	

(2 rows)

The contents of the field has become visible; we can now see the full list of privileges in there. Here is the adopted notation, which is sometimes not quite intuitive:

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

Column privileges are displayed separately (in the column with the corresponding name).

---

Now Bob's attempts are successful. To avoid specifying the schema name all the time, Bob adds it to the search path.

```
|| bob=> SET search_path = public, alice;
||
|| SET
||
|| bob=> UPDATE t1 SET n = n + 1;
||
|| UPDATE 1
||
|| bob=> SELECT * FROM t1;
||
||      n
||      ---
||      2
|| (1 row)
```

But other operations are still forbidden:

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

Bob can also access the first column of the t2 table:

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

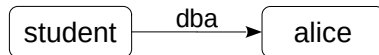
But it is forbidden to read the second column:

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

# Group Roles

## Including a role into a group

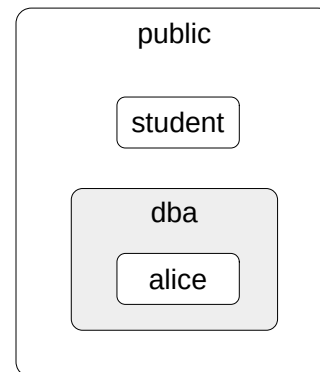
```
student=> GRANT dba TO alice;
```



the `public` pseudorole implicitly includes all the other roles

## Excluding a role from a group

```
student=> REVOKE dba FROM alice;
```



14

Apart from representing a database user, each role can also include other roles, i.e., represent a group of roles. A role can be included into another role, just like a Unix user can be included into a group.

It is also possible to include a group role into another group (but circular membership loops are not allowed). In fact, PostgreSQL does not differentiate between single-user and group roles.

The idea of such inclusion is to make group role privileges available to single-user 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. It facilitates database administration and access control.

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.

The following roles have the right to modify a role by including or excluding other roles:

- the role to be included (or excluded)
- a role with the `SUPERUSER` attribute
- a role with the `CREATEROLE` attribute

<https://postgrespro.com/docs/postgresql/15/role-membership>



# Predefined Group Roles

<code>pg_signal_backend</code>	— terminate sessions and cancel queries	} <code>pg_monitor</code>
<code>pg_read_all_settings</code>	— read server parameters	
<code>pg_read_all_stats</code>	— access statistics	
<code>pg_stat_scan_tables</code>	— access statistics that take locks	
<code>pg_read_all_data</code>	— read data in all tables	
<code>pg_write_all_data</code>	— modify data in all tables	
<code>pg_read_server_files</code>	— read files on the server	
<code>pg_write_server_files</code>	— write files on the server	
<code>pg_execute_server_programs</code>	— execute programs on the server	
<code>pg_checkpoint</code>	— execute CHECKPOINT command	

PostgreSQL provides a number of predefined roles that have some privileged capabilities. These roles can be granted by superusers to DBA roles, allowing them to perform specific tasks that otherwise would require SUPERUSER attribute.

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

<https://postgrespro.com/docs/postgresql/15/default-roles>

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

## Group Roles

Remember, Bob cannot read the second column of the table:

```
bob=> SELECT * FROM t2;

ERROR:  permission denied for table t2
```

Let’s grant pg\_read\_all\_data privilege to Bob. This can be done by a superuser:

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

GRANT ROLE

Now Bob can access all tables, as if he was granted SELECT on all tables and USAGE on all schemas:

```
bob=> SELECT * FROM t2;

  n | who
-----+-----
  1 | alice
100 | bob
(2 rows)
```

## Privileges of the public pseudorole

- connection to any database
- access to the public schema and the right to create objects in it
- access to the system catalog
- execution of any routines
- privileges are automatically granted on each new object

revoked as of  
PostgreSQL 15

## Configuring default privileges

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

17

As we have already said, the public pseudorole 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.
- Write access to the public schema (this was revoked in PostgreSQL 15).
- The right to execute 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 the required privileges on newly created objects. It can be also used to revoke the EXECUTE privilege from the public pseudorole.

<https://postgrespro.com/docs/postgresql/15/sql-alterdefaultprivileges>

## Default Privileges and Routines

Alice creates a function:

```
alice=> CREATE FUNCTION foo() RETURNS integer AS $$  
SELECT 1;  
$$ LANGUAGE sql IMMUTABLE;  
  
CREATE FUNCTION
```

Can Bob call it if Alice has not granted him the EXECUTE privilege?

```
|| bob=> SELECT foo();  
  
||  
||      foo  
||      ----  
||      1  
||      (1 row)
```

Yes, the call is possible. The EXECUTE privilege is automatically granted to the public role for each newly created function.

It can be avoided if you modify the default privileges:

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

Now the public role has no EXECUTE privilege for functions owned by Alice, so only Alice can run them.

```
alice=> CREATE FUNCTION bar() RETURNS integer AS $$  
SELECT 1;  
$$ LANGUAGE sql IMMUTABLE;  
  
CREATE FUNCTION  
  
|| bob=> SELECT bar();  
  
|| ERROR: permission denied for function bar
```

Policies complement privileges  
to manage access to tables at the row level

A policy is applied

- to particular roles

- to particular commands (SELECT, INSERT, UPDATE, DELETE)

A policy defines row access conditions

- permissive: allows row access if the condition is true

- restrictive: forbids row access if the condition is false

- different conditions (predicates) for existing rows and the rows to be inserted

While privileges provide access control at the table and column levels, row-level security policies do it at the row level.

By default, RLS is switched off. If required, it has to be enabled for each table explicitly.

Policies can be defined for a table, a set of commands (SELECT, INSERT, UPDATE, DELETE), and for particular roles. In fact, each policy is a boolean condition (a predicate), which is computed for each selected row. If the condition is true, the access is allowed (the access must not be forbidden by any *restrictive* policy and must be allowed by at least one *permissive* policy at the same time). Otherwise, the row won't be visible.

The predicates defining access to existing and newly added rows can differ (in this case, an UPDATE operation will be successful only if both predicates are true).

Predicates are computed with the rights of the caller.

RLS policies do not apply to the table owner (in most cases), superusers, roles with the BYPASSRLS attribute, and cannot be used for integrity constraints (uniqueness, foreign keys).

<https://postgrespro.com/docs/postgresql/15/ddl-rowsecurity>

## Row-Level Security

Security policies enable us to control table access at the row level, for each particular role.

```
|  alice=> SELECT * FROM t2;
```

```
|      n | who
|-----+-----
|      1 | alice
|     100 | bob
| (2 rows)
```

To see how it works, let's make the role reading the table see only its own rows, i.e., in which the who field contains this role's name.

```
|  alice=> CREATE POLICY who_policy ON t2
|  USING (who = current_user);
```

```
|  CREATE POLICY
```

For security policy to come into effect, it has to be enabled at the table level:

```
|  alice=> ALTER TABLE t2 ENABLE ROW LEVEL SECURITY;
```

```
|  ALTER TABLE
```

Now Bob sees only its own rows. In fact, each row has to be checked separately during query execution to verify that it satisfies the predicate specified in the policy.

```
||    bob=> SELECT n FROM t2;
```

```
||      n
||-----
||     100
|| (1 row)
```

```
||    bob=> INSERT INTO t2(n) VALUES (101);
```

```
||    INSERT 0 1
```

```
||    bob=> SELECT n FROM t2;
```

```
||      n
||-----
||     100
||     101
|| (2 rows)
```

Row-level security policies do not apply to superusers and roles with the BYPASSRLS attribute. The table owner usually bypasses these policies as well:

```
|  alice=> SELECT * FROM t2;
```

```
|      n | who
|-----+-----
|      1 | alice
|     100 | bob
|     101 | bob
| (3 rows)
```

But the owner can choose to limit its own rights:

```
|  alice=> ALTER TABLE t2 FORCE ROW LEVEL SECURITY;
```

```
|  ALTER TABLE
```

```
|  alice=> SELECT * FROM t2;
```

```
|      n | who
|-----+-----
|      1 | alice
| (1 row)
```

Roles, privileges, and policies provide a flexible access control mechanism for different usage scenarios

- you can easily allow everything to everyone
- you can set up strict access control if required

When creating a new role, it is important to ensure that it can connect to the server

1. Create a database, a schema, and a table with two columns: a key and a value.  
Create a role.
2. Find out the IP address and the subnet mask of the virtual machine and configure the system so that the connection from this address is allowed only to the created role and only to the created database, using password authentication.  
Make sure that `student` is still locally trusted by the server.
3. Configure access control so that the created role can query the table and change the values in it, but not the keys.
4. Restore original authentication settings.

**Task 2.** Make sure you have a backup copy of the `pg_hba.conf` file before making changes.

The IP address and the subnet mask can be found with the following command:

```
ip a | awk '/scope global/ {print $2}'
```

Do not forget about the `listen_addresses` parameter.



## Task 1. Create a database, objects, and a role

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

student=# \c access_overview
You are now connected to database "access_overview" as user "student".

student=# CREATE SCHEMA s;
CREATE SCHEMA

student=# CREATE TABLE s.t(
    key integer PRIMARY KEY,
    value text
);

CREATE TABLE

student=# INSERT INTO s.t VALUES (1,'One'),(2,'Two');
INSERT 0 2

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

## Task 2. Set up authentication

It's always a good idea to make a backup copy of a configuration file before modifications:

```
postgres$ cp -n /etc/postgresql/15/main/pg_hba.conf /etc/postgresql/15/main/pg_hba.conf.backup
```

Here is the IP address and the subnet mask:

```
postgres$ ip a | awk '/scope global/ {print $2}'
10.0.2.15/24
```

Rewrite the HBA file:

```
postgres$ echo 'host access_overview alice 10.0.2.15/24 scram-sha-256' > /etc/postgresql/15/main/pg_hba.conf
postgres$ echo 'local all student trust' >> /etc/postgresql/15/main/pg_hba.conf
```

Set the listen\_addresses parameter:

```
student=# SHOW listen_addresses;

listen_addresses
-----
localhost
(1 row)
```

```
student=# ALTER SYSTEM SET listen_addresses = '*';
```

ALTER SYSTEM

```
student$ sudo pg_ctlcluster 15 main restart
```

Test the changes.

Local connection is not allowed for alice:

```
student$ psql "host=localhost user=alice dbname=access_overview password=alicepass" -c "SELECT now();"
psql: error: connection to server at "localhost" (127.0.0.1), port 5432 failed: FATAL: no pg_hba.conf entry for host "127.0.0.1", user "alice", database "access_overview", SSL encrypt
```

connection to server at "localhost" (127.0.0.1), port 5432 failed: FATAL: no pg\_hba.conf entry for host "127.0.0.1", user "alice", database "access\_overview", no encryption

Connection to other databases is also not allowed:

```
student$ psql "host=10.0.2.15 user=alice dbname=student password=alicepass" -c "SELECT now();"
psql: error: connection to server at "10.0.2.15", port 5432 failed: FATAL: no pg_hba.conf entry for host "10.0.2.15", user "alice", database "student", SSL encryption
```

connection to server at "10.0.2.15", port 5432 failed: FATAL: no pg\_hba.conf entry for host "10.0.2.15", user "alice", database "student", no encryption

```
student$ psql "host=10.0.2.15 user=alice dbname=access_overview password=alicepass"
```

```
|  alice=> SELECT now();
|
|              now
|-----+-----
| 2023-05-04 19:51:33.285121+03
| (1 row)
```

## Task 3. Restrict access

```
student$ psql -d access_overview
```

```
student=# GRANT ALL ON SCHEMA s TO alice;
```

GRANT

```
student=# GRANT SELECT ON s.t TO alice;
```

GRANT

```
student=# GRANT UPDATE(value) ON s.t TO alice;
```

GRANT

Test the privileges:

```
|  alice=> SELECT * FROM s.t;
```

```
|
|  key | value
|-----+-----
|  1   | One
|  2   | Two
| (2 rows)
```

```
|  alice=> UPDATE s.t SET value = 'Unity' WHERE key = 1;
```

```
|  UPDATE 1
```

```
|  alice=> UPDATE s.t SET key = key+1 WHERE key = 2;
```

```
|  ERROR: permission denied for table t
```

Everything works as expected.

## Task 4. Revert changes

```
student=# ALTER SYSTEM RESET ALL;
```

ALTER SYSTEM

```
postgres$ cp /etc/postgresql/15/main/pg_hba.conf.backup /etc/postgresql/15/main/pg_hba.conf
```

