

Database: **dvdrental** · Admin user: **postgres** · Trainee user: **trainee**

Task Objectives

By completing this lab, students will learn how to:

1. Connect to PostgreSQL using the **psql** command-line tool.
2. Create a new user named **trainee**.
3. Grant permission to connect to the **dvdrental** database.
4. Grant **SELECT** permission on specific tables.
5. Log in as **trainee** and test permissions.
6. Update and revoke permissions using SQL commands.

Step 1: Open the PostgreSQL Command Line

Open Command Prompt (CMD) or a terminal and connect to PostgreSQL as the administrator.

```
psql -U postgres -d postgres
```

Enter the password for **postgres** when prompted.

If **psql** is not recognized, use the full path to **psql.exe** from your PostgreSQL installation folder, for example:

```
"C:\Program Files\PostgreSQL\17\bin\psql.exe" -U postgres -d postgres
```

Change **17** to your installed PostgreSQL version.

Verify the connection:

```
SELECT current_user, current_database();
```

Expected result:

```
current_user | current_database
-----+-----
postgres    | postgres
```

Step 2: Create the **trainee** User

Execute the following SQL command:

```
CREATE USER trainee
WITH PASSWORD 'Tr@1234';
```

Verify the user:

```
\du
```

The `\du` command lists PostgreSQL roles and their attributes.

Understand the code:

- `CREATE USER` creates a PostgreSQL role that can log in.
- `trainee` is the username.
- `WITH PASSWORD` sets the login password.

Step 3: Grant Permission to Connect to `dvdrental`

Run this command as `postgres`:

```
GRANT CONNECT  
ON DATABASE dvdrental  
TO trainee;
```

This allows the user to connect to the `dvdrental` database. It does not automatically allow the user to read tables.

Step 4: Exit and Log In as `trainee`

Exit the administrator session:

```
\q
```

Now connect as the new user:

```
psql -U trainee -d dvdrental -W
```

`-W` (uppercase): Forces psql to prompt for a password before connecting.

Enter the password:

```
Tr@1234
```

Verify the login:

```
SELECT current_user, current_database();
```

Expected result:

```
current_user | current_database
-----+-----
trainee      | dvdrental
```

If this works, the user has successfully connected to the database.

Step 5: Test Table Access Before Granting **SELECT**

As **trainee**, execute:

```
SELECT *
FROM public.film
LIMIT 5;
```

Expected result: A permission-denied error if **trainee** has not already been granted access.

PostgreSQL manages database, schema, and table permissions separately.

Step 6: Grant **SELECT** Permission

Exit the trainee session:

```
\q
```

Reconnect as the administrator:

```
psql -U postgres -d postgres
```

connect with dvdrental database:

```
\c dvdrental
```

Grant access to the **film** table:

```
GRANT SELECT
ON TABLE dvdrental.film
TO trainee;
```

the table permission allows the user to read data from **film**.

Step 7: Test the Permission Again

Exit the administrator session and log in as **trainee**:

```
psql -U trainee -d dvdrental -W
```

Run:

```
SELECT film_id, title, release_year  
FROM public.film  
LIMIT 10;
```

Expected result: Up to 10 film records.

You can also check which user is running the query:

```
SELECT current_user;
```

Step 8: Test Restricted Permissions

While logged in as **trainee**, try the following queries.

Test A: Read the **actor** table

```
SELECT *  
FROM public.actor  
LIMIT 5;
```

Expected: permission denied if **trainee** has not been granted **SELECT** on **actor**.

Test B: Insert a record

```
INSERT INTO public.film (title, language_id)  
VALUES ('Student Practice Film', 1);
```

Expected: permission denied because **INSERT** has not been granted.

Test C: Update a record

```
UPDATE public.film  
SET title = 'Updated Test Film'  
WHERE film_id = 1;
```

Expected: permission denied because **UPDATE** has not been granted.

If a query succeeds unexpectedly, check for permissions inherited from other roles or earlier grants.

Step 9: Update Permissions

Log in as **postgres** again.

```
psql -U postgres -d postgres
```

Task A: Allow reading the **actor** table

```
GRANT SELECT
ON TABLE public.actor
TO trainee;
```

Test as **trainee**:

```
SELECT *
FROM public.actor
LIMIT 5;
```

Task B: Allow inserting film records

```
GRANT INSERT
ON TABLE public.film
TO trainee;
```

The standard **dvdrental** database uses a generated **film_id**, so test with:

```
INSERT INTO public.film (title, language_id)
VALUES ('Student Practice Film', 1);
```

Task C: Allow updating film records

```
GRANT UPDATE
ON TABLE public.film
TO trainee;
```

Test with:

```
UPDATE public.film
SET title = 'Updated Test Film'
WHERE film_id = 1;
```

Task D: Revoke update permission

```
REVOKE UPDATE
ON TABLE public.film
FROM trainee;
```

Test the **UPDATE** statement again as **trainee**. It should fail if no other permission grants update access.

Step 10: View and Verify Permissions

As the administrator, inspect the table privileges:

```
\dp public.film
```

This displays the access privileges for the **film** table.

To list database privileges:

```
\l dvdrental
```

To list users and roles:

```
\du
```

To check whether the user can connect:

```
SELECT has_database_privilege(
    'trainee',
    'dvdrental',
    'CONNECT'
);
```

Expected result:

```
t
```

To check whether the user can read the **film** table:

```
SELECT has_table_privilege(
    'trainee',
    'public.film',
    'SELECT'
);
```

Expected result: **t** if the privilege is granted, otherwise **f**.

Student Practice Tasks

1. Create a user named trainee2

```
CREATE USER trainee2 WITH PASSWORD 'Practice@123';
```

2. Grant database connection

```
GRANT CONNECT ON DATABASE dvdrental TO trainee2;
```

4. Grant SELECT on customer

```
GRANT SELECT ON TABLE public.customer TO trainee2;
```

5. Test access to payment

```
SELECT * FROM public.payment LIMIT 5;
```

6. Grant SELECT on payment

```
GRANT SELECT ON TABLE public.payment TO trainee2;
```

7. Revoke SELECT on payment

```
REVOKE SELECT ON TABLE public.payment FROM trainee2;
```

Important: Run administrative **CREATE USER**, **GRANT**, and **REVOKE** commands as **postgres** or another suitably privileged role. Run the testing queries while logged in as the relevant trainee user. For the practice tasks, grant and revoke commands must be executed in the appropriate session; they are not all intended to run as **trainee**.

Lab Questions

1. What is the purpose of **psql**?
2. What is the difference between **CONNECT** and **SELECT** permissions?
3. Why can a user connect to a database but still be unable to read a table?
4. What is the purpose of **GRANT USAGE ON SCHEMA public**?
5. What is the difference between **GRANT** and **REVOKE**?
6. What does **has_table_privilege()** return?
7. How can you verify a user's permissions using **\dp** and **\du**?

Lab conclusion: Students should be able to create PostgreSQL users, manage database and table permissions, log in through **psql**, and verify access by running SQL queries.