Overview of Database Security

Objectives

After watching this video, you will be able to:

- Describe different levels of database security
- Explain the difference between authentication and authorization
- Identify the granularity of security on objects in a database
- Describe how auditing, encryption, and application security can enhance your database security

Levels of database security

Server security

- On-premise servers:
  - Who has access?
  - How are they physically secured?
- Managed cloud:
  - Check provider documentation

Operating system configuration

- Regular patching
  - Is it regularly patched with the latest tested updates
- System hardening
  - Uses a known configuration to reduce vulnerabilities
- Access monitoring
  - Is continuously monitored for any unauthorized access

RDBMS configuration

similar to OS

- Regular patching
- Review and use system-specific security features
- Reduce the number of administrators
  - Small set of trusted users have administrative privileges

Accessing databases and objects

Authentication

- Similar to:
  - PIN for cell phone
  - Password for computer
- Verifies that the user is who they claim to be:
  - For example, by validating username and password
- External authentication methods
  - PAM
  - Windows login IDs
  - LDAP
  - Kerberos

Authorization

- Authorized to access:
  - Objects
  - Data
- Grant privileges to:
  - Users
  - Groups
  - Roles

Privileges

In the table, we can allow users to select, insert, update or delete data.

product_id	product	category_id
1	Rich coffee	100
2	Smooth coffee	100
3	Breakfast tea	101
4	Earl grey tea	101
5	Assam tea	101

Auditing

- Monitor:
  - Who accesses what objects
  - What actions they perform
- Audit:
  - Actual access against security plan

Encryption

- Adds another layer of security:
  - Intruders need to decrypt
- Industry & regional regulations:
  - Algorithms
  - Key management
- Performance impact

more assets, more secure, more used resources and time

Application security

- SQL injection strings
- Insecure code

Users, Groups, and Roles

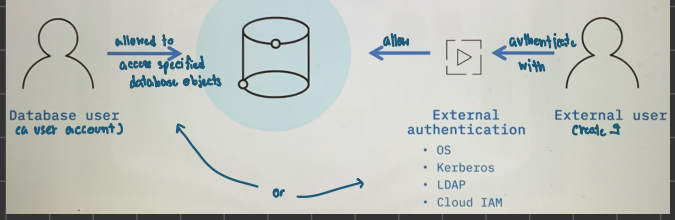
Objectives

After watching this video, you will be able to:

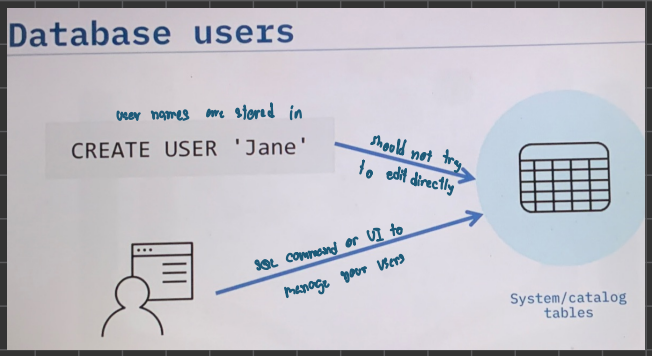
- Describe users, groups, and roles
- Explain how users, groups, and roles work together
- Manage database security objects

Database users

In most systems, users need to be explicitly granted access to database objects



Database users



Database groups

useful when you are using that OS to authenticate your users

Ex. PostgreSQL you define group in database

SQL server or Db2

OR

Logical grouping of users

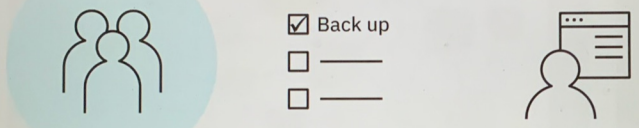
Mapped to OS group

Database roles

→ similar to database group that they contain privileges and access right to all users of that role

to define permission needed to specific role

↳ EX, a backup operator role would have permissions to access a database and perform backup functions



Database roles

Predefined roles:

- databaseowner
- backupoperator
- datareader
- datawriter

Custom roles:

- salesperson
- accountsclerk
- groupheads
- developer
- tester

Managing security objects

that need to the same functionality to fulfill their job role

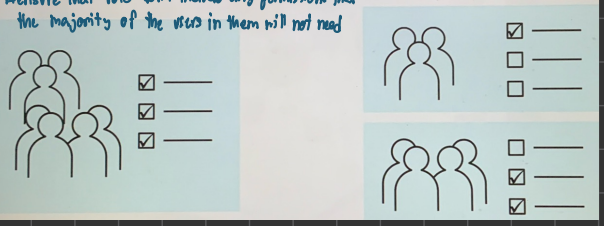
Assigning privileges to groups or roles simplifies security management ↳ if that job changes in the future

- Adding permissions } quick, easier and less prone to mistake rather than individual users
- Adding new users
  ↳ if new users join a team
- One user can be a member of one or more groups or roles

Managing security objects

Principle of least privilege

↳ ensure that role don't include any permissions that the majority of the users in them will not need



Managing Access to Databases and Their Objects

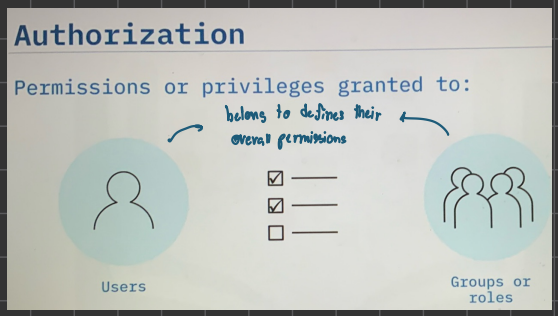
Objectives

After watching this video, you will be able to:

- Grant access to objects in a database
- Revoke access to objects in a database
- Deny access to objects in a database

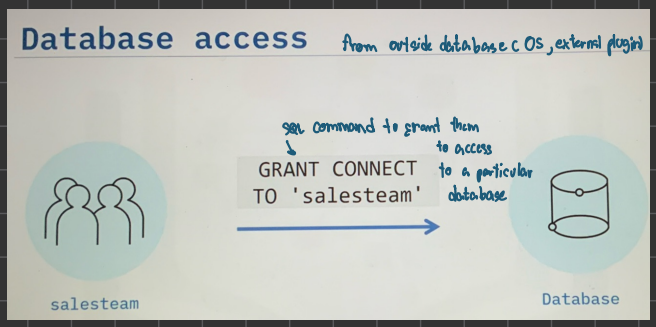
Authorization

Permissions or privileges granted to:

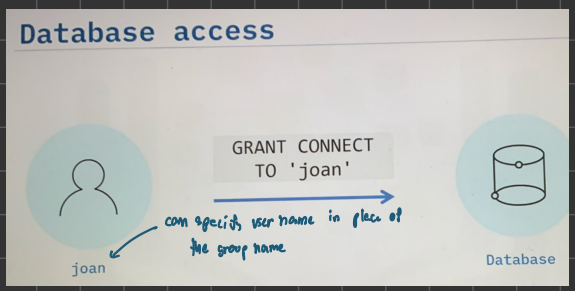


Database access

from outside database c OS, external plugin



Database access



Database access

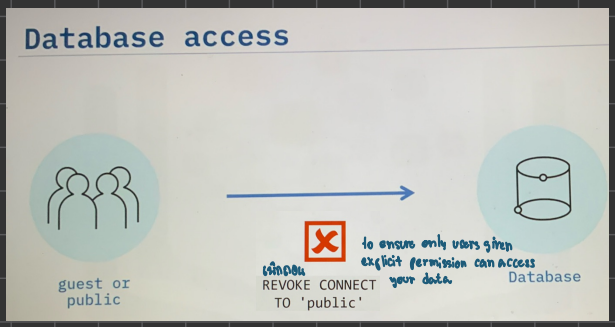
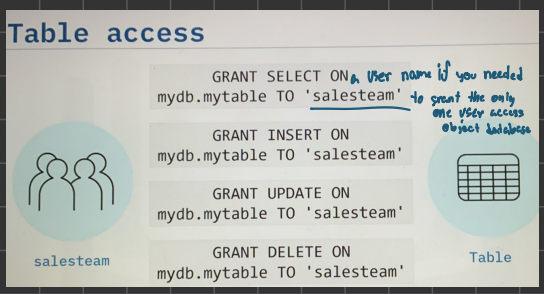
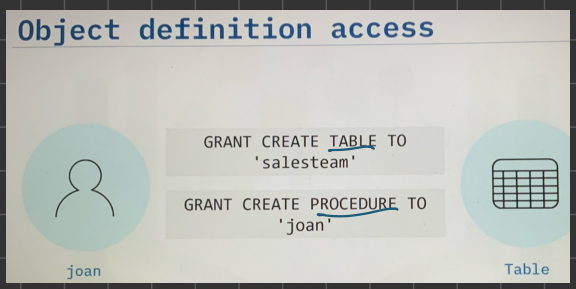


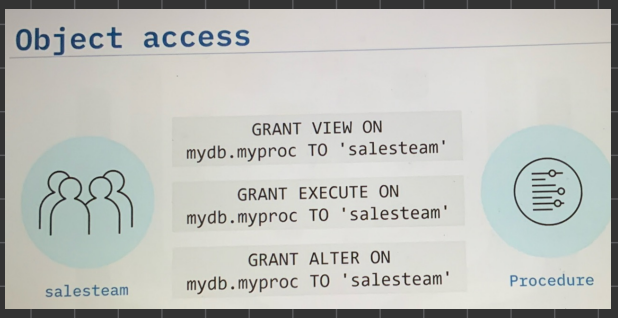
Table access



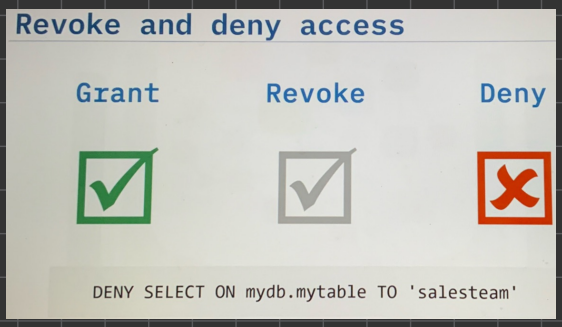
Object definition access



Object access



Revoke and deny access



Auditing Database Activity

Objectives

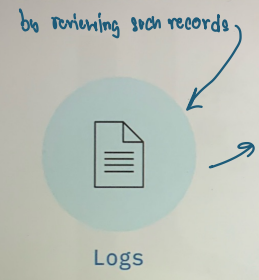
After watching this video, you will be able to:

- Explain why auditing is important
- Describe how to audit database access
- Describe how to audit database activity

Why audit your database?

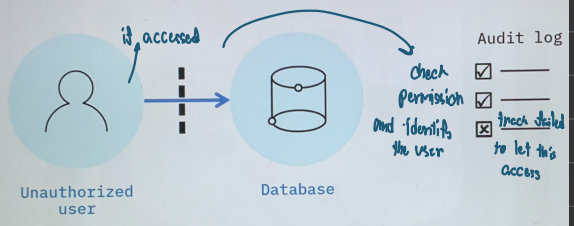
- Does not directly offer protection
- But it does help you to:
  - Identify gaps in security system
  - Track errors in privilege administration
- Compliance
- Implement on premise and in the cloud

Why audit your database?



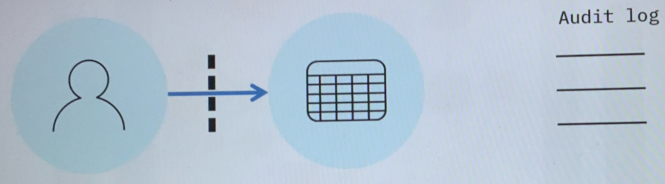
- Identify suspicious activity
- Respond to security threats

Auditing database access



- Db2 on Cloud – built-in audit facility
- MySQL – audit log

Auditing database activity



- Db2 on Cloud – change history event monitor
- PostgreSQL – pgAudit

Encrypting Data

Objectives

After watching this video, you will be able to:

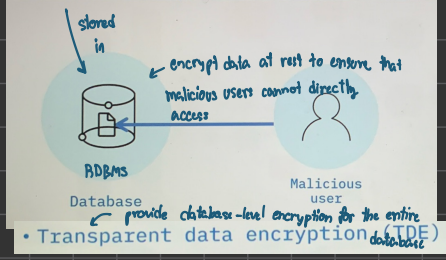
- Explain the advantages of encrypting data
- Describe encryption algorithms and keys
- Identify the differences between symmetric and asymmetric encryption
- Explain transparent data encryption
- Explain customer managed keys
- List encryption vs. performance trade-offs
- Describe how RDBMSs can encrypt data in transit

Why encrypt your data?

- Another layer in security system
- Often last line of defense
- Can protect data:
  - At rest
  - During transmission
- May be required by:
  - Industry
  - Region
  - Customer

(some industries (financial and medical services))

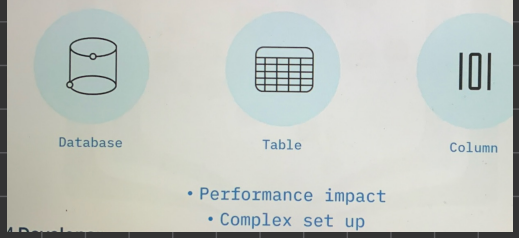
Protecting data at rest



- Transparent data encryption (TDE)

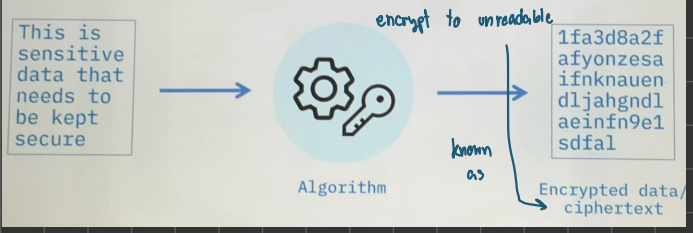
(provide database-level encryption for the entire database)

Protecting data at rest



- Performance impact
- Complex set up

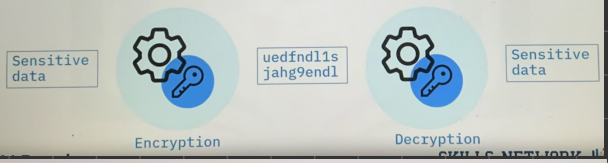
Algorithms and keys



Symmetric encryption

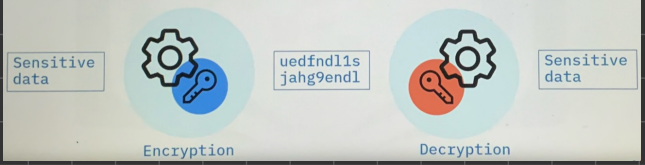
(key pair)

- Same key to encrypt and decrypt
- Examples: AES, DES
- Key is shared with all users
- Increased likelihood of compromise



Asymmetric encryption

- Uses two keys: one public, one private
- Public key encrypts, private key decrypts
- Examples: RSA, ECC
- Must store private keys securely

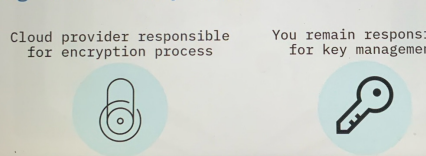


Transparent data encryption

- Encryption and key management
- Not visible to users
- Database engine encrypts and decrypts data
- Also encrypts backups

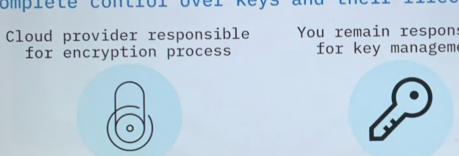
Customer managed keys

- Provide the data owner with more control over their data stored in the cloud
- Bring Your Own Key (BYOK) (protocol)



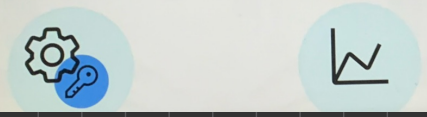
Customer managed keys

- Benefits:
  - Cloud provider cannot access your confidential data
  - Security admins manage keys; database admins manage data
  - Complete control over keys and their lifecycle



Encryption vs. performance

- Choice of symmetric or asymmetric encryption may be configurable
- All encryption takes time and effort
- Asymmetric algorithms generally use longer keys, therefore have greater overheads
- Symmetric algorithms are often sufficient



Protecting data in transit

- Often provided by the RDBMS
- Protocols:
  - TLS
  - SSL
- May encrypt by default, may be configurable
- Performance impact

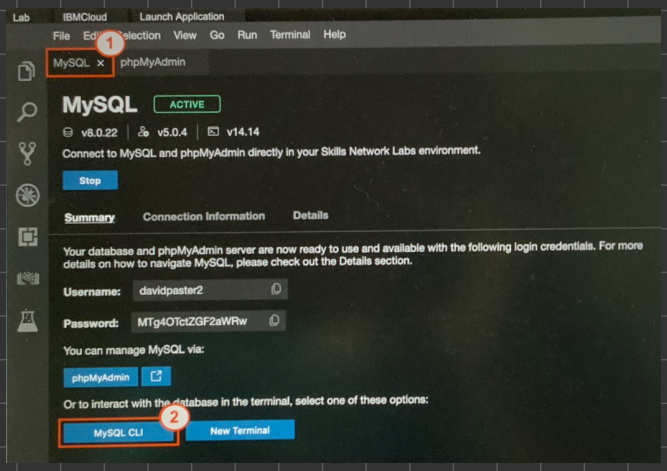
LAB

Exercise 3: Secure data using encryption

In this example exercise, you will learn how to secure your data adding extra layer of security using data encryption. There may be certain parts of your database containing sensitive information which *should not* be stored in plain text. This is where encryption comes in.

You will implement encryption and decryption of a column in the world database using the official AES (Advanced Encryption Standard) algorithm. AES is a symmetric encryption where the same key is used to encrypt and decrypt the data. The AES standard permits various key lengths. By default, key length of 128-bits is used. Key lengths of 196 or 256 bits can be used. The key length is a trade off between performance and security. Let's get started.

1. Click the **MySQL CLI** button from the mysql service session tab.



2. First, you will need to hash your passphrase (consider your passphrase is **My secret passphrase**) with a specific hash length (consider your hash length is **512**) using a hash function (here you will use hash function from **SHA-2** family). It is good practice to hash the passphrase you use, since storing the passphrase in plaintext is a significant security vulnerability. Use the following command in the terminal to use the SHA2 algorithm to hash your passphrase and assign it to the variable `key_str`:

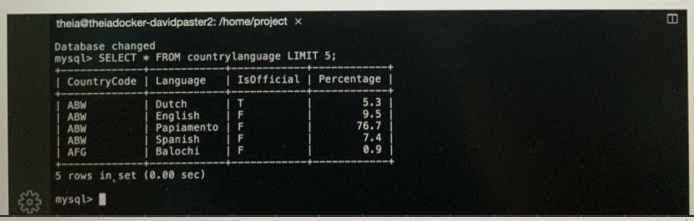
```
SET @key_str = SHA2('My secret passphrase', 512);
```

3. Now, let's take a look at the `world` database. First, you will want to connect to the database by entering the following command in the CLI:

```
USE world;
```

4. Next, let's take a quick look at the `countrylanguage` table in our database with the following command:

```
SELECT * FROM countrylanguage LIMIT 5;
```



For *demonstration purposes*, **suppose** that the last column in the table, labeled *Percentage* contains sensitive data, such as a citizen's passport number. Storing such sensitive data in plain text is an enormous security concern, so let's go ahead and encrypt that column.

5. To encrypt the *Percentage* column, we will first want to convert the data in the column into binary byte strings of length 255 by entering the following command into the CLI:

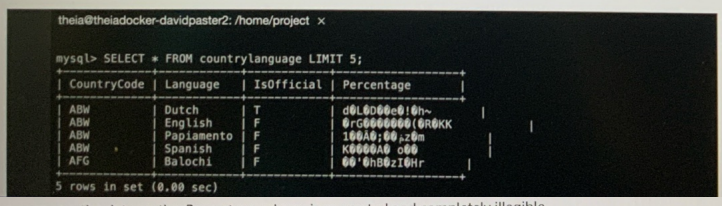
```
ALTER TABLE countrylanguage MODIFY COLUMN Percentage varbinary(255);
```

6. Now to actually encrypt the *Percentage* column, we use the AES encryption standard and our hashed passphrase to execute the following command:

```
UPDATE countrylanguage SET Percentage = AES_ENCRYPT(Percentage, @key_str);
```

7. Let's go ahead and see if the column was successfully encrypted by taking another look at the *countrylanguage* table. We again run the same command as in step 4:

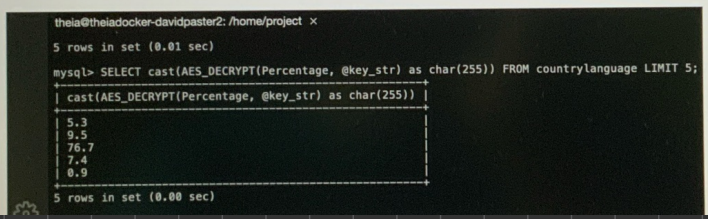
```
SELECT * FROM countrylanguage LIMIT 5;
```



As you can see, the data on the *Percentage* column is encrypted and completely illegible.

8. The supposedly sensitive data is now encrypted and secured from prying eyes. However, we should still have a way to access the encrypted data when needed. To do this, we use the `AES_DECRYPT` command, and since AES is symmetric, we use the same key for both encryption and decryption. In our case, recall that the key was a passphrase which was hashed and stored in the variable `key_str`. Suppose we need to access the sensitive data in that column. We can do so by entering the following command in the CLI:

```
SELECT cast(AES_DECRYPT(Percentage, @key_str) as char(255)) FROM countrylanguage;
```



LAB

Exercise 1: Create New Roles and Grant them Relevant Privileges

In PostgreSQL, users, groups, and roles are all the same entity, with the difference being that users can log in by default.

In this exercise, you will create two new roles: `read_only` and `read_write`, then grant them the relevant privileges.

To begin, ensure that you have the PostgreSQL Command Line Interface open and connected to the `demo` database, as such:

```
theia@theiadocker-davidpastern: /home/project          theia@theiadocker-davidpastern: /home/project

ALTER DATABASE
ALTER DATABASE
demo=# \dt
                List of relations
  Schema   |       Name        | Type  |  Owner
-----------+-------------------+-------+----------
 bookings | aircrafts_data    | table | postgres
 bookings | airports_data     | table | postgres
 bookings | boarding_passes   | table | postgres
 bookings | bookings          | table | postgres
 bookings | flights           | table | postgres
 bookings | seats             | table | postgres
 bookings | ticket_flights    | table | postgres
 bookings | tickets           | table | postgres
(8 rows)

demo=#
```

Task A: Create a `read_only` role and grant it privileges

1. To create a new role named `read_only`, enter the following command into the CLI:

```
CREATE ROLE read_only;
```

2. First, this role needs the privilege to connect to the `demo` database itself. To grant this privilege, enter the following command into the CLI:

```
GRANT CONNECT ON DATABASE demo TO read_only;
```

3. Next, the role needs to be able to use the schema in use in this database. In our example, this is the `bookings` schema. Grant the privilege for the `read_only` role to use the schema by entering the following:

```
GRANT USAGE ON SCHEMA bookings TO read_only;
```

4. To access the information in tables in a database, the `SELECT` command is used. For the `read_only` role, we want it to be able to access the contents of the database but not to edit or alter it. So for this role, only the `SELECT` privilege is needed. To grant this privilege, enter the following command:

```
GRANT SELECT ON ALL TABLES IN SCHEMA bookings TO read_only;
```

This allows the `read_only` role to execute the `SELECT` command on **all** tables in the bookings schema.

Task B: Create a `read_write` role and grant it privileges

1. Similarly, create a new role called `read_write` with the following command in the PostgreSQL CLI:

```
CREATE ROLE read_write;
```

2. As in Task A, this role should first be given the privileges to connect to the `demo` database. Grant this privilege by entering the following command:

```
GRANT CONNECT ON DATABASE demo TO read_write;
```

3. Give the role the privileges to use the `bookings` schema that is used in the `demo` database with the following:

```
GRANT USAGE ON SCHEMA bookings TO read_write;
```

4. So far the commands for the `read_write` role have been essentially the same as for the `read_only` role. However, the `read_write` role should have the privileges to not only access the contents of the database, but also to create, delete, and modify entries. The corresponding commands for these actions are `SELECT`, `INSERT`, `DELETE`, and `UPDATE`, respectively. Grant this role these privileges by entering the following command into the CLI:

```
GRANT SELECT, INSERT, DELETE, UPDATE ON ALL TABLES IN SCHEMA bookings TO read_write;
```

Exercise 2: Add a New User and Assign them a Relevant Role

In this exercise, you will create a new user for the database and assign them the one of the roles you created in Exercise 1. This method streamlines the process of adding new users to the database since you don't have to go through the process of granting custom privileges to each one. Instead, you can assign them a role and the user inherits the privileges of that role.

Suppose you wish to add a new user, `user_a`, for use by an information and help desk at an airport. In this case, assume that there is no need for this user to modify the contents of the database. As you may have guessed, the appropriate role to assign is the `read_only` role.

1. To create a new user named `user_a`, enter the following command into the PostgreSQL CLI:

```
CREATE USER user_a WITH PASSWORD 'user_a_password';
```

In practice, you would enter a secure password in place of `'user_a_password'`, which will be used to access the database through this user.

2. Next, assign `user_a` the `read_only` role by executing the following command in the CLI:

```
GRANT read_only TO user_a;
```

3. You can list all the roles and users by typing the following command:

```
\du
```

You will see the following output:

```
postgres=# \du
                                   List of roles
 Role name |                         Attributes                         | Member of
-----------+------------------------------------------------------------+-----------
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
 read_only | Cannot login                                               | {}
 user_a    |                                                            | {read_only}
```

Notice that `user_a` was successfully created and that it is a member of `read_only`.

Exercise 3: Revoke and Deny Access

In this exercise, you will learn how to revoke a user's privilege to access specific tables in a database.

Suppose there is no need for the information and help desk at the airport to access information stored in the `aircrafts_data` table. In this exercise, you will revoke the `SELECT` privilege on the `aircrafts_data` table in the `demo` database from `user_a`.

1. You can use the `REVOKE` command in the Command Line Interface to remove specific privileges from a role or user in PostgreSQL. Enter the following command into the PostgreSQL CLI to remove the privileges to access the `aircrafts_data` table from `user_a`:

```
REVOKE SELECT ON aircrafts_data FROM user_a;
```

2. Now suppose `user_a` is transferred departments within the airport and no longer needs to be able to access the `demo` database at all. You can remove all their `SELECT` privileges by simply revoking the `read_only` role you assigned to them earlier. You can do this by entering the following command in the CLI:

```
REVOKE read_only FROM user_a;
```

3. Now you can check all the users and their roles again to see that the `read_only` role was successfully revoked from `user_a` by entering the following command again:

```
\du
```

You will see the following output:

```
demo=# REVOKE read_only FROM user_a;
REVOKE ROLE
demo=# \du
                                   List of roles
 Role name |                         Attributes                         | Member of
-----------+------------------------------------------------------------+-----------
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
 read_only | Cannot login                                               | {}
 user_a    |                                                            | {}
```

Notice that `user_a` is still present but it is no longer a member of the `read_only` role.