BUT Science des données Aurillac

SAE 2.08 : Rapport

Projet RBAC PostgreSQL

Réalisé par :
Moustapha Ndiaye
Abdallah Remmide

Enseignant :
Kevin Atighehchi

Année universitaire 2024/2025

UNIVERSITÉ Clermont Auvergne

Introduction

Dans le cadre du projet de la SAE 2.08 portant sur la mise en place d'un contrôle d'accès RBAC (Role-Based Access Control), nous devions travailler sur une base de données de gestion d'un hôpital. L'objectif était d'adapter un script SQL initialement conçu pour Oracle afin qu'il soit compatible avec PostgreSQL, et également mettre en œuvre un système de rôles et de permissions montrant l'organisation d'un établissement hospitalier. Voici les différentes étapes que nous avons suivies.

2) Choix de la base de données

Nous avons choisi le deuxième lien (BD de gestion d'un hôpital) :

https://www.mycompiler.io/view/3i8r3XzgaAQ

3) Identification des rôles et hiérarchisation

Après avoir analysé la structure de la base de données hospitalière, nous avons identifié plusieurs rôles liés aux différents métiers présents dans un hôpital. Voici les rôles retenus, accompagnés de leurs principaux accès :

admin: a accès à toutes les tables de la base de données, sans restriction

doctor: peut consulter et modifier les informations des patients, diagnostics, rendez-vous et examens

nurse: a accès en lecture aux patients, et peut également enregistrer des soins et diagnostics

receptionist: gère les rendez-vous et les informations de contact

pharmacist: peut accéder aux médicaments et gérer les achats

Nous avons également établi une hiérarchie des rôles pour simplifier la gestion des permissions. Le rôle admin hérite des droits de tous les autres rôles, tandis que les autres (doctor, nurse, etc.) sont indépendants les uns des autres.

4) Installation de PostgreSQL

Nous avons installé PostgreSQL sur nos machines afin de pouvoir exécuter et tester les scripts SQL

5) Modifications apportées sur le script initial

Nous avons modifié plusieurs éléments du script de base pour qu'il fonctionne correctement sur PostgreSQL :

1)
Les types de données spécifiques à Oracle comme VARCHAR2 ont été remplacés par VARCHAR, qui est standard et accepté par PostgreSQL. Le type NUMBER a été remplacé par INTEGER pour les identifiants et NUMERIC(6,3) pour les champs contenant des prix.

2)
PostgreSQL étant sensible à la casse, nous avons renommé toutes les tables et les attributs en minuscules (ex : hospital, doc_id, etc.) pour éviter les erreurs.

3)
Les guillemets doubles ont été remplacés par des guillemets simples pour les chaînes de caractères. Les valeurs NULL ont été écrites en majuscules. Les dates ont été mises au format ISO (YYYY-MM-DD), comme exigé par PostgreSQL.

4)
Les instructions INSERT INTO ont été réorganisées pour respecter les contraintes de clés étrangères. Par exemple, les hôpitaux ont été insérés avant les docteurs, les docteurs avant les infirmiers, etc..

5)
Certains noms contenaient des fautes ou des incohérences. Nous les avons corrigés pour assurer l'uniformité, comme "Salmaniya Hospital" corrigé en "Salamaniya Hospital".

6)
Les lignes contenant des identifiants non définis ont été corrigées ou replacées au bon endroit, comme par exemple les INSERT INTO qui tentaient d'utiliser un doc_id ou un nurse_id non inséré avant. Les valeurs comme (NULL, 26) ont été conservées si elles respectaient les contraintes de la base.

Grâce à ces modifications, le script fonctionne sur PostgreSQL. En effet, toutes les données sont insérées avec succès et les requêtes s'exécutent sans erreur.

6) Mise en place du système RBAC

Pour sécuriser l'accès à la base de données, nous avons mis en place un système RBAC. Cette approche permet d'attribuer des permissions en fonction du rôle de l'utilisateur plutôt que de les gérer individuellement :

a)

Création des comptes utilisateurs
Nous avons créé un utilisateur pour chaque type de profil :
utilisateur_medecin, utilisateur_nurse, utilisateur_reception, utilisateur_pharmacien.

b)

Création des rôles
Chaque utilisateur a un rôle dédié :
role_medecin, role_nurse, role_reception, role_pharmacien.

c)

Attribution des rôles
Chaque utilisateur a reçu le rôle correspondant à son métier. Par exemple, utilisateur_nurse a été lié au rôle role_nurse.

d)

Rôle global
Nous avons aussi créé un rôle général role_hopital, qui regroupe tous les autres. Cela permet, par exemple, de donner un accès complet à un administrateur.

e)

Attribution des privilèges
Chaque rôle a reçu uniquement les permissions suivantes :

-Le médecin peut lire et insérer dans diagnosis et examine

-L'infirmier peut gérer les patients

-Le réceptionniste gère les rendez-vous

-Le pharmacien gère les achats

Ce système permet de limiter les droits selon les besoins de chaque utilisateur, garantissant la sécurité et la confidentialité des données.

7) Scénario d’accès à la BD

Dans un hôpital connecté à la base de données abdallahremmide, différents professionnels de santé utilisent le système au quotidien. Voici un scénario typique montrant l’accès à la base depuis différentes sessions, selon le rôle de chaque utilisateur.

Le docteur Qasim

Le docteur Qasim termine la consultation d’un patient et souhaite enregistrer un nouveau diagnostic. Il se connecte à la base avec son compte à l’aide du mot de passe “medecin1”

```
Last login: Sat May 17 22:39:39 on ttys000
abdallahremmide@Mac ~ % psql -U utilisateur_medecin -d abdallahremmide
Password for user utilisateur_medecin:
```

L’accès lui a été autorisé :

```
Last login: Sat May 17 22:39:39 on ttys000
abdallahremmide@Mac ~ % psql -U utilisateur_medecin -d abdallahremmide
Password for user utilisateur_medecin:
psql (17.4 (Postgres.app))
Type "help" for help.

abdallahremmide=>
```

Il ajoute ensuite le diagnostic avec succès :

```
abdallahremmide=> INSERT INTO diagnosis (diagnos_no, issue_date, treatment, remarks, nurse
_id, doc_id)
abdallahremmide-> VALUES (42, '2025-05-17', 'Check-up complet', 'RAS', 2, 7);
INSERT 0 1
abdallahremmide=>
```

Il tente ensuite d’afficher les patients, mais cela lui a été refusé :

```
abdallahremmide=> SELECT * FROM patient;
ERROR:  permission denied for table patient
abdallahremmide=>
```

“ERROR : permission denied for table patient”. Cela confirme que le médecin ne peut pas accéder aux données des patients directement, ce qui protège la confidentialité. Il quitte ensuite la session à l’aide de “\q”.

```
abdallahremmide=> \q
abdallahremmide@Mac ~ %
```

L’infirmière Zainab

Zainab, infirmière de l’hôpital, vient de prendre en charge une nouvelle patiente : Imane Saleh. Elle se connecte à l’aide du mot de passe “nurse2”.

```
abdallahremmide@Mac ~ % psql -U utilisateur_nurse -d abdallahremmide
Password for user utilisateur_nurse:
```

L’accès lui a été autorisé :

```
abdallahremmide@Mac ~ % psql -U utilisateur_nurse -d abdallahremmide
Password for user utilisateur_nurse:
psql (17.4 (Postgres.app))
Type "help" for help.
```

Elle ajoute maintenant la patiente :

```
abdallahremmide=> INSERT INTO patient (ssn, fname, lname, age, gender, nurse_id, rec_id)
abdallahremmide-> VALUES (100000011, 'Imane', 'Saleh', 38, 'F', 7, 41);
INSERT 0 1
abdallahremmide=>
```

L’insertion est autorisée. Elle quitte ensuite la session

```
abdallahremmide=> \q
abdallahremmide@Mac ~ %
```

Le réceptionniste Ahmed

Le réceptionniste Ahmed prend un rendez-vous pour Imane. Il se connecte à l’aide du mot de passe “reception3”

```
abdallahremmide@Mac ~ % psql -U utilisateur_reception -d abdallahremmide
Password for user utilisateur_reception:
```

L’accès lui a été autorisé :

```
abdallahremmide@Mac ~ % psql -U utilisateur_reception -d abdallahremmide
Password for user utilisateur_reception:
psql (17.4 (Postgres.app))
Type "help" for help.

abdallahremmide=>
```

Il prend ensuite le rendez-vous et exécute :

```
abdallahremmide=> INSERT INTO appointment (appoint_no, appoint_date, appoint_time, rec_id)

abdallahremmide-> VALUES (11, '2025-05-18', '10:00:00', 41);
INSERT 0 1
abdallahremmide=>
```

C’est un succès. Il tente ensuite de voir les diagnostics :

```
abdallahremmide=> SELECT * FROM diagnosis;
ERROR:  permission denied for table diagnosis
abdallahremmide=>
```

C’est un échec. L’accès lui a été refusé, il n’a pas la permission.

La pharmacienne Fatima

Fatima, pharmacienne, ajoute un achat de médicament pour la patiente Imane. Dans un premier temps, elle se connecte à l’aide du mode de passe “pharmacien4”.

```
abdallahremmide@Mac ~ % psql -U utilisateur_pharmacien -d abdallahremmide
Password for user utilisateur_pharmacien:
```

L’accès lui a été autorisé.

```
abdallahremmide@Mac ~ % psql -U utilisateur_pharmacien -d abdallahremmide
Password for user utilisateur_pharmacien:
psql (17.4 (Postgres.app))
Type "help" for help.

abdallahremmide=>
```

Puis elle ajoute l’achat de médicament :

```
abdallahremmide=> INSERT INTO purchase (ssn, reg_no)
abdallahremmide-> VALUES (100000011, 28);
INSERT 0 1
abdallahremmide=>
```

L’achat est bien enregistré. Elle essaie ensuite de voir tous les rendez-vous :

```
abdallahremmide=> SELECT * FROM appointment;
ERROR:  permission denied for table appointment
abdallahremmide=>
```

On peut voir que c’est refusé car son rôle n’inclut pas les droits sur les rendez-vous.

Ce scénario montre bien que le système de gestion des droits par rôles (BRAC) fonctionne correctement. Chaque utilisateur s’est connecté à la base avec son propre compte. Cela confirme que les privilèges sont bien appliqués selon le rôle de chaque utilisateur. La base de données est sécurisée et respecte la séparation des responsabilités.

Conclusion

Pour conclure, tout au long de ce projet nous avons travaillé sur un script initialement conçu pour Oracle que nous avons adapté à PostgreSQL. Nous avons modifié les types de données, l'ordre des insertions, les noms des tables et les formats de date pour que le script fonctionne. Nous avons ensuite mis en place un système de gestion des utilisateurs et des rôles avec des privilèges bien définis. Chaque utilisateur possède uniquement les droits dont il a besoin selon son métier. Le scénario final a permis de tester et de prouver que la sécurité des accès est bien respectée. Ce travail nous a appris à structurer une base de données, à sécuriser les accès et à simuler un environnement multi-utilisateurs réaliste et fonctionnel.