Apprendre POSTGRES

Après installation de **POSTGRES** on doit se connecter à **postgres** avec la commande:

```
sudo -i -u postgres
```

La commande psql

Cette commande est utilisé pour vous connecter à un utilisateur et une base de données de votre serveur.

Dès la connexion faite, vous aurez à votre disposition un terminal PostgresSQL pour envoyer des commandes SQL à votre serveur.

Par abus, les gens se suffisent simplement d'entrer la commande suivante pour se connecter

Si vous êtes connecté à l'utilisateur toto, cette commande sera équivalente à la commande suivante :

```
psql --username=ndeko --dbname=ndeko
```

Le premier `ndeko` est l'utilisateur auquel se connecter et le second est la base de données à laquelle se connecter.

Nous vous conseillons d'utiliser la version complète pour toujours êtes sûr de l'utilisateur et la base à laquelle vous êtes connectée.

Pour simplement exécuter une commande dans une base de données

La commande psql nous permet également de simplement exécuter une commande SQL dans une base de données sans s'y connecter :

Quitter le terminal

Pour quitter le terminal PostgreSQL, il suffit d'entrer `\q.`

Création d'un utilisateur PostgreSQL

Cette commande est fondamentale donc il est nécessaire de bien comprendre son principe. Il faut tout d'abord avoir en tête qu'il n'a aucun lien obligatoire entre l'utilisateur Unix et l'utilisateur PostgreSQL **même** si leur nom est le même.

Pour créer un utilisateur, la démarche est très simple accédé au shell de l'utilisateur Linux
nommé postgres et entrer la commande suivante :

```
createuser ndeko
```

Pour jeter un œil à la liste des utilisateurs, il faudra vous connecter au shell PostgreSQL via la commande `psql`
puis entrer la commande suivante.

```
SELECT rolname FROM pg_roles;

#attention au ; vous n'etes pas dans le shell linux en SQL les commandes finissent tou
jours avec ;
```

Suppression d'un utilisateur

Pour supprimer un utilisateur, il suffit d'entrer dans le shell Linux la commande suivante :

```
dropuser ndeko
```

Création d'une base de données et concept d'owner

Pour créer une base de donnée on entre simplement la commande suivante dans le shell Linux :

```
createdb ndekodb
```

Dans ce cas, l'*owner ou le propriétaire de la Base de données* sera l'utilisateur **postgres**.

Prenons le temps d'expliquer ce concept, l'*owner* (propriétaire) a tous les droits sur sa base de données. Il pourra créer, modifier et détruire les tables de sa base de données.

Il est bien évidemment possible de créer la base de données avec un *owner* particulier, **mais** il faut se connecter au shell PostgreSQL via la commande `psql` :

Il est bien évidemment possible de créer la base de données avec un *owner* particulier, **mais** il faut se connecter au shell PostgreSQL via la commande `psql` :

```
CREATE DATABASE testdb OWNER test;
```

Il est également possible de changer le propriétaire avec la commande suivante :

```
ALTER DATABASE testdb OWNER TO ndeko;
```

Notons que la commande ALTER permet de modifier beaucoup plus que le propriétaire d'une base de données.

Donner/retirer des droits particuliers

Donner/retirer des droits particuliers

Vous pouvez, pour chaque rôle, gérer les droits sur les bases de données

Attention à la syntaxe !

Lorsque vous n'êtes pas dans le shell Linux, il est nécessaire d'adopter la syntaxe SQL et donc de bien finir chaque commande par un " **;**".

```
GRANT CREATE ON DATABASE testdb TO marc; #Attribue le droit de création
```

```
REVOKE CREATE ON DATABASE testdb FROM marc; #Retire le droit de création
```

On vient d'attribuer les droits de création d'objets à *marc*
dans la base de données `testdb`
. Il existe bien d'autres droits que vous trouverez sans problème selon vos besoins dans la documentation (ALTER, DROP, INSERT, UPDATE, ...).

Pour faciliter les manipulations lorsque le nombre d'utilisateurs devient grand, il est possible "d'attribuer un rôle à un autre". Le second héritera dès lors des droits du premier :

```
GRANT marc TO test;
```

Exercices : Gestion des droits

Premiers pas

Je suis connecté en tant que `postgres` sur Linux et souhaite créer une base de données nommée `joe` pour un utilisateur encore inexistant nommé également `joe`.

Question

Quelles commandes taper pour créer un utilisateur et une base de données nommés joe?

Solution

```
createuser joe
createdb joe
```

Question

Comment faire pour que la base de données ait pour *owner* (propriétaire) joe ?

Solution

On se connecte d'abord au shell PostgreSQL avec la commande : `psql -U postgres postgres` [on accède en fait à la base nommée postgres via l'utilisateur postgres].

Puis, on exécute la commande SQL suivante :

```
ALTER DATABASE joe OWNER TO joe;
```

Retirer tous les droits

On souhaite retirer tous les droits de joe sur la base de données nommée testdb

Question

Comment faire ?

Indice

Quelle commande fait l'inverse de GRANT ?

Solution

```
REVOKE ALL ON DATABASE testdb FROM joe;
```

Héritage (Lire le complément sur la différence entre rôle et utilisateur avant)

Il existe un rôle nommé admin ayant la possibilité de créer et modifier les tables sur toutes les bases de données du SGBD.

Question

Comment faire pour que l'utilisateur Joe puisse accéder aux mêmes droits ?

```
GRANT admin TO joe;
```

Différence utilisateur et rôle dans PostgreSQL

Concept de rôles dans PostgreSQL

Dans ce SGBD, ce principe de rôles permet à la fois de créer des utilisateurs et des groupes qui seront vus de la même manière par le système. Notons néanmoins que les notions d'utilisateurs et de groupes ne sont pas aussi usuels que dans un système Unix.

La création d'un utilisateur se confondra alors avec une création de rôle.

Création d'un utilisateur PostgreSQL

Cette commande est fondamentale donc il est nécessaire de bien comprendre son principe. Il faut tout d'abord avoir en tête qu'il n'a aucun lien obligatoire entre

l'utilisateur Unix et l'utilisateur PostgreSQL **même** si leur nom est le même.

Lançons d'abord la commande : `psql` pour accéder au shell PostgreSQL (Unix traduira ça en "`psql -U postgres postgres`", soit une connexion à la base de données postgres via l'utilisateur postgres).

Il existe deux commandes SQL qui amèneront à un résultat sensiblement équivalent :

```
CREATE USER test;
```

```
CREATE ROLE test; #il faut simplement ajouter l option LOGIN pour obtenir l équivalenc
e avec CREATE USER
```

Pour simplifier les démarches, certains systèmes d'exploitation proposent des commandes directement dans le bash:

```
createuser test
```

Commandes ayant des équivalences dans le shell Unix et dans le shell PostgreSQL

Il existe quelques commandes Unix ayant des équivalences dans le shell PostgreSQL. Les plus importantes sont `createdb`, `createuser`, `dropuser` et `dropdb`.

La forme shell Unix est la plus rapide et pratique. L'intérêt d'utiliser la forme SQL est d'avoir accès à un nombre d'options plus grand. Par exemple, la forme shell Unix de création de base de données ne permet pas de choisir un utilisateur particulier en tant que *owner.*

Attention à la syntaxe !

Lorsque vous n'êtes pas dans le shell, il est nécessaire d'adopter la syntaxe SQL et ainsi de bien finir chaque commande par un " `;` ".

Maintenant que nous pouvons créer des utilisateurs dans la base de données, il nous faut une méthode permettant de vérifier si un utilisateur qui tente de se connecter dispose d'identifiants valides. Implémentons donc notre fonction `login`:

```
/**
   * @description Pour Connecter les utilisateurs existant
   * @param {Request} req La requete de l'utilisateur
   * @param {Response} res La reponse à envoyer  l'utilisateur
   * @param {Function} next La methode pour passer au middleware suivant
   * @author NdekoCode
   * @memberof UserController
   */
  login(req, res, next) {
    const alert = new Alert(req, res);
    if (!isEmpty(req.body)) {
      const errors = {
        ...ValidateEmail(req.body.email),
        ...validPassword(req.body.password),
      };
      if (isEmptyObject(errors)) {
        return userModel
          .findOne({ email: req.body.email })
          .then((user) => {
            // On verifie si l'utilisateur a été trouver
            if (isEmpty(user)) {
              // Si l'utilisateur n'a pas été trouver on envois une reponse 401
              return alert.danger("Email ou mot de passe incorrect");
            }
            // L'utilisateur veut se connecter en meme temps on essaie de vefifier son
identité en comporant si le mot de passe qu'il entrer est issue du meme mot de passe q
ui se trouve dans la base de donnée
            compare(req.body.password, user.password)
              .then((valid) => {
                // On verifie si il y a une erreur d'authentification
                if (!valid) {
                  // L'utilisateur n'existe pas alors on envois une erreur arbitraire
                  return alert.danger("Email ou mot de passe incorrect");
                }
                // L'utilisateur existe on envois alors son ID et un token d'authentif
ication
                return alert.makeAlert("Vous etes connecter", 200, "success", {
                  userId: user._id,
                  token: "TOKEN",
                });
              })
              .catch((error) =>
                alert.danger(
                  "Erreur survenus lors de la connexion de l'utilisateur",
                  500
                )
              );
          })
          .catch(() => alert.danger("Email ou mot de passe incorrect", 500));
      }

      return alert.danger("Email ou mot de passe invalide");
    }
    return alert.danger("Entrer des informations valides");
  }
```

Dans cette fonction :

- Nous utilisons notre modèle Mongoose pour vérifier que l'e-mail entré par l'utilisateur correspond à un utilisateur existant de la base de données :
  - Dans le cas contraire, nous renvoyons une `erreur 401 Unauthorized`.
  - Si l'e-mail correspond à un utilisateur existant, nous continuons.
- Nous utilisons la fonction `compare` de `bcrypt` pour comparer le mot de passe entré par l'utilisateur avec le hash enregistré dans la base de données :
  - S'ils ne correspondent pas, nous renvoyons une `erreur 401 Unauthorized` avec le même message que lorsque l'utilisateur n'a pas été trouvé, afin de ne pas laisser quelqu'un vérifier si une autre personne est inscrite sur notre site.
  - S'ils correspondent, les informations d'identification de notre utilisateur sont valides. Dans ce cas, nous renvoyons une réponse `200` contenant l'ID utilisateur et un *token*.

  • La méthode `compare` de bcrypt compare un string avec un hash pour, par exemple, vérifier si un mot de passe entré par l'utilisateur correspond à un hash sécurisé enregistré en base de données. Cela montre que même bcrypt ne peut pas décrypter ses propres hashs.

Créez des tokens d'authentification

Comment créer des tokens d'authentification