Secteur Tertiaire Informatique

Filière « Etude et développement »

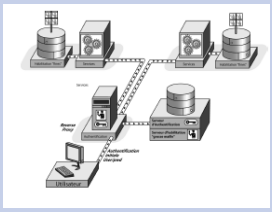
TP – Fonctions et procédures stockées

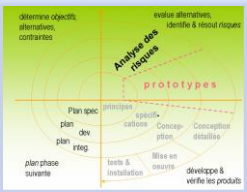
Persistance de données

Apprentissage

Mise en situation

Evaluation





1. DESCRIPTIF DU SYSTEME

1.1 CAHIER DES CHARGES

L'entreprise COGIP a besoin d'un outil pour aider ses approvisionneurs logistiques à gérer le suivi des commandes de consommables.

Les consommables peuvent être de tout ordre, par exemple :

- Papier pré-imprimé (pour les commandes, factures, fiches de paie, ...) ;
- Papeterie diverse (ramettes de feuilles A4, stylos, agrafes…) ;
- Cartouches d'encre, toners
- …

Pour chacun de ces produits, il existe plusieurs fournisseurs possibles ayant déjà livré la société.

En plus de ces fournisseurs, de nombreux représentants viennent prospecter l'entreprise. Il est important de conserver les coordonnées des commerciaux pour d'éventuelles commande ou appels d'offre futurs.

La procédure d'approvisionnement s'effectue de la façon suivante :

1. L'entreprise lance un appel d'offre qui se matérialise par un envoi d'email aux fournisseurs susceptibles de faire une offre valable.

   Cet email précise la nature de la demande (type de consommable, quantité prévisible de la commande, quantité annuelle, délai de livraison courant, délai de livraison en cas de rupture de stock).
2. Les fournisseurs intéressés par le marché renvoient leurs une proposition commerciale à l'approvisionneur afin qu'il puisse faire un choix.
3. Par suite de la décision de l'approvisionneur logistique, des commandes peuvent être envoyées au fournisseur pour l'achat d'un ou de plusieurs produits pour une quantité donnée. Cette quantité peut être livrée en plusieurs fois.

Les seules informations mémorisées sur la livraison sont la date de dernière livraison ainsi que la quantité livrée totale.

Une base de données nommée « cogip-supply » est fournie, ses documents descriptifs sont indiqués dans la suite de ce document.

1.2 MODELE DE DONNEES

Ci-dessous le modèle **logique de données** (aussi appelé « diagramme d'entité-relation ») présentant la base de données mise en place pour répondre au besoin du projet.

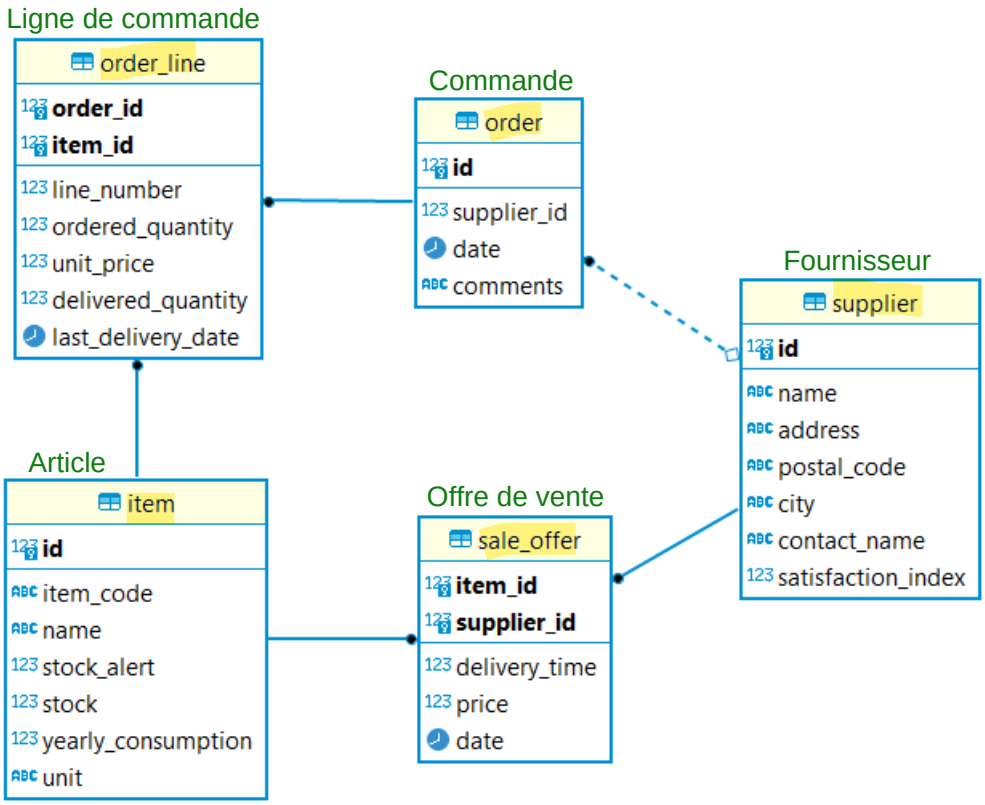


Figure 1 : Modèle logique de données (aka. diagramme d'entité-relation)

1.3 DICTIONNAIRE DE DONNEES

Ci-dessous vous trouverez un détail des données stockées en BDD.

1.3.1 Table « supplier »

Contient les informations concernant les fournisseurs.

Attribut	Description	Type
id		int
name	~~Contact chez le fournisseur~~ Nom F	varchar(15)
Address	~~Date de commande~~ Addresse	~~datetime~~
Postal_code	~~Délai de livraison~~ Code Postal	varchar(5)
City	~~Date dernière livraison~~ Nom ville	~~datetime~~ varchar(25)
Contact_name	~~Libellé Produit~~ Contact chez le fournisseur	varchar(30)
Satisfaction_index	Indice de satisfaction entre, valeur entre 1 et 10 (ou null)	int

Contraintes :

- Tous les champs sont obligatoires, sauf « satisfaction_index »
- Le code postal est constitué de 5 chiffres
- L'indice de satisfaction est compris dans une échelle de 1 à 10

1.3.2 Table « item »

Contient les informations concernant les produits commandés par l'entreprise COGIP.

Attribut	Description	Type
id		int
item_code	Code de l'article suivant une nomenclature spécifique.	char(4)
name	Nom complet de l'article.	datetime
stock_alert	Quantité minimum en stock permettant de déclencher une alerte.	int
stock	La quantité d'articles en stock.	Int
yearly_consumption	Consommation annuelle estimée.	int
unit	Indique la quantité d'éléments de l'article (par exemple, une ramette de 1000 pages A4 aura sa colonne unit à « BP1000 »	Varchar(15)

Contraintes :

- Tous les champs sont obligatoires (non nuls)

1.3.3 Table « sale_offer »

Détails les informations d'une proposition commerciale effectuée par un fournisseur.

Attribut	Description	Type
item_id	Clef étrangère vers un enregistrement de la table « item ».	
supplier_id	Clef étrangère vers un enregistrement de la table « supplier ».	
delivery_time	Délai de livraison estimé (en jour)	int
price	Prix unitaire proposé.	int
date	Date de la proposition commerciale.	datetime
stock	La quantité d'articles en stock.	Int

Contraintes :

- Tous les champs sont obligatoires
- Le code produit doit exister dans la table « item »
- Le code fournisseur doit exister dans la table « supplier »

1.3.4 Table « order »

Contient les informations d'une commande effectuée auprès d'un fournisseur.

Attribut	Description	Type
id		
date	Date de la commande	Datetime
comments	Commentaires portant sur la commande.	varchar(800)

Contraintes :

- Tous les champs sont obligatoires sauf « comments » ;
- L'identifiant est un champ auto-incrémenté ;
- La date de commande est par défaut la date du jour lors de la création de la commande.

1.3.5 Table « order_line »

Contient des informations sur la commande d'un article (une commande est composée de plusieurs « lignes de commande », une par article).

Attribut	Description	Type
order_id	Clef étrangère vers un enregistrement de « order ».	Int
item_d	Clef étrangère vers un enregistrement de « item ».	int
ordered_quantity	Le nombre d'articles commandés.	int
delivered_quantity	Le nombre d'articles livrés.	Int
Unit_price	Le prix unitaire pour cette commande (n'est pas forcément similaire au prix unitaire de la proposition commerciale).	float
Last_delivery_date	Date de la dernière livraison.	datetime

Contraintes :

- Tous les champs sont obligatoires, sauf la quantité livrée (qui évoluera en fonction des livraisons) ;
- L'identifiant de la commande doit exister dans la table « order » (clef étrangère)
- L'identifiant de l'article doit exister dans la table « item » (clef étrangère)

2. CREATION DE PROCEDURES/FONCTIONS

Tous les SGBD offrent la possibilité de créer des procédures ou fonctions stockées.

La différence entre fonction et procédure est la suivante : une procédure **n'a pas de type de retour (elle ne renvoie rien)**.

Comme dans tout autre langage, les procédures/fonction sont des ensembles d'instructions écrites dans un langage procédural et impératif propre au SGBD. Ces fonctions sont précompilées et stockées en BDD puis exécutées sur demande. Système de Gestion de Base de Données

Les avantages des fonctions et procédures stockées sont multiples :

- Meilleures **performances** de requêtage dû à la compilation ;
- Meilleure **sécurité** → il est possible de forcer l'utilisateur de la base de données à passer par les procédures stocker pour l'empêcher d'accéder aux tables de la base de données via des requêtes traditionnelles.

2.1 LANGAGE PL/PGSQL

La PL/pgSQL est l'acronyme de « **Procedural Language / Postgre Structured Query Language** » est une extension du langage SQL et permet de créer des traitements procéduraux sur les bases de données.

On retrouve des langages similaires disponibles pour une très grande majorité des systèmes de gestion de base de données (T-SQL pour les bases de données SQL server de Microsof, PL/SQL pour les bases de données de l'entreprise Oracle).

Une bonne documentation du langage PL/pgSQL est disponible à l'adresse suivante : https://www.postgresqltutorial.com/postgresql-plpgsql/

2.2 DEFINITION D'UNE FONCTION

Une fonction doit être pré-créée en utilisant un ordre de création de fonction tel que :

```
CREATE OR REPLACE FUNCTION <nom-fonction>(<paramètres>)
 RETURNS <type-de-retour>
 LANGUAGE plpgsql -- langage procédural utilisé, ici plpgsql
AS $$ -- "AS" indique le début de la définiton de la fonction
declare  -- la partie "declare" permet de déclarer toutes les variables utilisées
dans le block délimité par "BEGIN-END"
    <nom-variable> <type-variable>;
begin
    -- ici le code de la fonction
    -- ce code doit retourner une valeur en accord avec le type de retour de la
fonction
END;
$$
```

Déclaration de variables :

Toutes les variables utilisées par la fonction doivent être définies dans la partie « declare ».

Les types utilisables sont les types PostgreSQL que vous commencez à connaître (voici une liste complète des types disponibles : https://docs.postgresql.fr/9.6/datatype.html).

Dans la suite de cet atelier vous écrirez un ensemble de fonctions qui seront ajoutées à la base de données « cogip-supply » qui vous a été fournie

2.3 FONCTION « FORMAT_DATE »

Cette partie permet de créer une première fonction de formatage de date

2.3.1 Descriptif

Objectif de la fonction :

Transformer en information de type « date » en une chaîne de caractère. Le format de la date devra suivre une représentation telle que :

« DD< séparateur-choisi >MM<séparateur-choisi>YYYY<séparateur-choisi> ».

Avec :

- DD → le jour sur 2 chiffres
- MM → le mois sur 2 chiffres
- YYYY → l'année sur 4 chiffres

Le séparateur entre les différents éléments sera un paramètre de la fonction.

Donnée d'entrée :

- date : la date à traiter
- separator : le séparateur sous forme de chaîne de caractères

Donnée de sortie :

- une chaîne de caractère représentant la date formatée

Jeu d'essai :

- Données d'entrée :
  - date : « 2021-01-15 15:42:07.000 »
  - separator : « / »
- Donnée de sortie :
  - « 15/01/2021 »

2.3.2 Code à tester

Voici le code permettant de créer une telle fonction.

```
CREATE OR REPLACE FUNCTION format_date(date date, separator varchar)
 RETURNS text
 LANGUAGE plpgsql
AS $$
begin
    -- en plpgsql, l'opérateur de concaténation est ||
    return to_char(date, 'DD' || separator || 'MM' || separator || 'YYYY');
END;
$$
```

1. **Analysez le code de la fonction et essayez de comprendre chacun de ses éléments.**

La présence des doubles dollar « $$ » est un mécanisme particulier propre à PostgreSQL, vous pourrez trouver une explication de son utilisation à l'adresse suivante : https://www.postgresqltutorial.com/postgresql-plpgsql/plpgsql-block-structure/

Le fonctionnement de la fonction « to_char » est détaillée ici.

Une fois la fonction créée dans votre base de données, vous pouvez l'utiliser en effectuant une requête « SELECT » et en y intégrant un appel.

Par exemple :

```
select format_date('2023-02-01', '/');
```

2. **Utilisez la nouvelle fonction dans une requête permettant d'afficher toutes les commandes avec un tiret ('-') utilisé en tant que séparateur.**

2.4 FONCTION « GET_ITEM_COUNT »

2.4.1 Descriptif

Objectif de la fonction :

La fonction doit afficher et retourner le nombre d'articles total en base de données. Elle utilisera une requête SQL pour récupérer le nombre d'articles.

Donnée d'entrée :

- Aucune

Donnée de sortie :

- Entier, le nombre d'articles de la base

2.4.2 Code à tester

Voici le code de la fonction à tester :

```
CREATE OR REPLACE FUNCTION get_items_count()
 RETURNS integer
 LANGUAGE plpgsql
AS $$
declare
    items_count integer;
    time_now time = now();
begin
    select count(id)
    into items_count
    from item;

    raise notice '% articles à %', items_count, time_now;

    return items_count;
END;
$$
```

3. **Analysez et testez le code, comment est effectuée l'affectation de la variable « items_count » ?**

2.5 FONCTION « COUNT_ITEMS_TO_ORDER »

Objectif de la fonction :

La fonction doit afficher (via un message « notice) et retourner le nombre d'articles pour lesquels la valeur du stock est inférieure au stock d'alerte.

Donnée d'entrée :

- Aucune

Donnée de sortie :

- Entier, le nombre d'articles avec un stock en alerte

4. **Implémentez une fonction qui répond au besoin.**

2.6 FONCTION « BEST_SUPPLIER »

Objectif de la fonction :

La fonction doit afficher le nom du fournisseur pour lequel il y a le plus eu de commandes effectuées (le plus d'enregistrements dans la table « order ») et retourner son identifiant.

Donnée d'entrée :

- Aucune

Donnée de sortie :

- Entier, l'identifiant du fournisseur pour lequel il y a le plus eu de commandes.

5. Implémentez une fonction qui répond au besoin.

2.7 FONCTION « SATISFACTION_STRING »

2.7.1 Descriptif

Objectif de la fonction :

Il y a dans la table « supplier », une colonne nommée « satisfaction_index » prenant une valeur de 1 à 10 ou Null.

Cet indice est mis à jour selon le bon vouloir de l'utilisateur (ceci via une interface graphique).

La fonction « satisfaction_string » devra pouvoir être utilisée pour retrouver une chaîne de caractère représentant l'indice de satisfaction.

Ci-dessous la correspondance entre un indice et la chaîne de caractères correspondante :

- **Indice = Null, 'Sans commentaire'**
- **Indice = 1 ou 2, 'Mauvais'**
- **Indice = 3 ou 4, 'Passable'**
- **Indice = 5 ou 6, 'Moyen'**
- **Indice = 7 ou 8, 'Bon'**
- **Indice = 9 ou 10, 'Excellent'**

Donnée d'entrée :

- **satisfaction_index :** entier, l'indice de satisfaction souhaité

Données de sortie :

- chaîne de caractères représentant l'indice de satisfaction en chaîne de caractères

2.7.2 Code à écrire

En tant que langage impératif, PL/pgSQL fournit toutes les structures de contrôle propres au langage impératif (« if », « while », « for »).

Pour cet exercice vous pourrez utiliser un ensemble de « if » ou un « switch-case » :

- Fonctionnement du « if » : https://www.postgresqltutorial.com/postgresql-plpgsql/plpgsql-if-else-statements/
- Fonctionnement du « switch-case » : https://www.postgresqltutorial.com/postgresql-tutorial/postgresql-case/

6. **Proposez deux fonctions, une basée sur un « if » et une autre sur un « switch-case ». Elles porteront les noms « satisfaction_string_if » et « satisfaction_string_case ».**
7. **Testez vos fonctions, en affichant le niveau de satisfaction des fournisseurs en toutes lettres ainsi que leur identifiant et leur nom grâce à une requête « SELECT ».**

Le résultat attendu est présenté par le tableau ci-dessous.

	id	name	satisfaction
1	120	GROBRIGAN	Bon
2	540	ECLIPSE	Bon
3	8700	MEDICIS	Sans commentaire
4	9120	DICOBOL	Bon
5	9150	DEPANPAP	Moyen
6	9180	HURRYTAPE	Sans commentaire

2.8 FONCTION « ADD_DAYS »

2.8.1 Descriptif

Objectif de la fonction :

« add_days » devra prendre en paramètre une date et un nombre de jours et retourner une nouvelle date incrémentée du nombre de jours.

Données d'entrée :

- **date :** date, une date à traiter
- **days_to_add :** entier, le nombre de jour à ajouter à la date

Donnée de sortie :

- une nouvelle date correspondant à la « date » + nombre de jours

Jeu d'essai :

- Données d'entrée :
  - Date : 2023-10-10
  - days_to_add : 5
- Donnée de sortie :
  - 2023-10-15

2.8.2 Code à écrire

Ajout de jours à une date :

Pour ajouter des jours à une date, inspirez-vous de la méthode présentée ici.

8. Créez la fonction « add_days ».

2.9 FONCTION « COUNT_ITEMS_BY_SUPPLIER »

2.9.1 Descriptif

L'objectif et de créer une fonction retournant le résultat d'une requête comptant le nombre d'articles proposés par un fournisseur.

Données d'entrée :

- **supplier_id :** entier, identifiant d'un fournisseur

Donnée de sortie :

- entier, le nombre d'articles proposés par une fournisseur

2.9.2 Code à écrire

Etape 1 : comptez les articles pour un fournisseur

Avant de vous lancer dans l'écriture de votre fonction, écrivez et testez votre requête qui compte les articles.

Pour compter les articles vendus par un fournisseur vous pourrez utiliser la table « **sale_offer** ».

Etape 2 : écrivez votre fonction

Pensez au paramètre à utiliser, au type de retour et aux variables.

Etape 4 : testez votre fonction

Etape 3 : ajoutez une erreur en cas de mauvais identifiant

Faites évoluer votre fonction en y ajoutant **une exception si l'identifiant passé en paramètre n'est utilisé dans aucun enregistrement** de la table « **supplier** ».

→ Coment vérifier si un enregistrement n'est associé à l'identifiant en paramètre ?

Voici un extrait de code PL/pgSQL permettant de vérifier si un enregistrement existe :

```
Declare – variable utilisée dans la fonction
    supplier_exists boolean;
begin -- début du corps de la fonction
    supplier_exists = exists(select * from supplier s where s.id = supplier_id);
    if supplier_exists = false then
        --  instructions si le fournisseur n'existe pas
    end if;
end;
```

```

```

→ Comment lever (ou « jeter ») une exception ?

Il est possible de lever une exception pour arrêter un une fonction (principe similaire à Java).

Pour se faire, vous pouvez adapter le code suivant :

```
RAISE EXCEPTION 'L''identifiant % n''existe pas', <identifiant> USING HINT =
'Vérifiez l''identifiant du fournisseur.';
```

Pour plus d'informations concernant le système d'exception vous pourrez vous référer à la documentation officielle : https://www.postgresql.org/docs/current/plpgsql-errors-and-messages.html

2.10 FONCTION « SALES_REVENUE »

2.10.1 Descriptif

« **sales_revenue** » devra calculer le chiffre d'affaires d'un fournisseur pour une année donnée en paramètre.

Attention :
on considérera que le chiffre d'affaires est stocké **hors taxe.** Le taux de TVA à utiliser est de **20,00%**.

Données d'entrée :

- **supplier_id :** entier, l'identifiant du fournisseur
- **year :** entier, l'année à considérer

Données de sortie :

1. **réel,** le chiffre d'affaire du fournisseur

2.10.2 Code à écrire

9. **Créez la fonction « sales_revenue », qui en fonction d'un identifiant fournisseur et d'une année entrée en paramètre, restituera le chiffre d'affaires de ce fournisseur pour l'année souhaitée.**

Dans votre fonction, il vous faudra utiliser une requête permettant de calculer le chiffre d'affaires (les tables « order » et « order_line » vous permettront d'arriver à vos fins).

2.11 FONCTION « GET_ITEMS_STOCK_ALERT »

2.11.1 Descriptif

Parmi les types de retour disponibles, il est possible de faire en sorte que la fonction **renvoie un ensemble d'enregistrements (similaire à une nouvelle table).**

Vous allez écrire une fonction nommée « get_items_stock_alert » qui renvoie l'identifiant (id), le code article (item_code), le nom (name) et la différence entre le stock d'alerte et le stock réel pour tous les articles qui ont leur valeur de stock inférieure à la valeur d'alerte.

La table ci-dessous présente le résultat attendu :

	id	item_code	name	stock_difference
1	1	B002	Bande magnétique 6250	8
2	3	D050	CD R-W 80mm	46
3	5	I105	Papier 2 ex continu	70
4	9	P230	Pre-imprime facture	250
5	13	R080	ruban Epson 850	8

2.11.2 Code à écrire

Ci-dessous une ébauche de la fonction à développer. Il vous faudra la compléter pour réussir à la faire fonctionner :

```
DROP FUNCTION get_items_stock_alert();
create or replace function get_items_stock_alert()
    returns table (
        id int,
        item_code character(4),
        -- TODO à compléter
    )
    language plpgsql
as $$
begin
    return query
        select
            -- TODO à compléter
        from
            item i
        where
            -- TODO à compléter
end;$$
```

10. Complétez la fonction afin qu'elle puisse fonctionner comme attendu.

2.12 PROCEDURE DE CREATION D'UTILISATEUR

2.12.1 Descriptif

Vous allez créer une procédure (fonction sans type de retour) qui pour objectif d'insérer un utilisateur en base de données (permet de sécuriser l'insertion des données).

Avant toute chose, il vous faut créer une nouvelle table « user » qui va stocker les informations de connexion d'un utilisateur.

Créez une nouvelle table ayant le schéma suivant :

`user(id, email, last_connection, password, role)`

Suivant le dictionnaire de données suivant :

Attribut	Description	Type	Contraintes
id		serial	
email		varchar	Non null
last_login	Date et heure de la dernière connexion.	datetime	
password	Mot de passe de l'utilisateur haché en utilisant « sha-1 ».	varchar	Non null
role	Rôle de l'utilisateur : « MAIN_ADMIN », « ADMIN », « COMMON »	varchar	Non null
connexion_attempt	Nombre de tentatives de connexion.	integer	0 par défaut
blocked_account	Indique si le compte est bloqué.	boolean	False par défaut

La procédure attendue devra effectuer un « insert » et utiliser ses paramètres pour remplir les colonnes.

Données d'entrées :

- Email, chaine de caractères
- Password (non haché), chaîne de caractères
- role, chaîne de caractères

L'intérêt d'une telle procédure est de pouvoir faire des vérifications sur les données en entrées. Si les données ne conviennent pas il faudra que la procédure lève une exception (en PL/pgSQL il vous faut utiliser « raise exception »).

Les règles de vérification que vous allez implémenter sont les suivantes :

- mot de passe de minimum 8 caractères
- email correctement formaté
- role dans la liste des rôles acceptables « MAIN_ADMIN », « ADMIN », « COMMON »

2.12.2 Code à écrire

Implémentez la fonction effectuant un « insert ». Avant cette requête vérifiez les paramètres de procédure.

Pour la vérification de la longueur du mot de passe vous pourrez utiliser la méthode « length() » : https://w3resource.com/PostgreSQL/length-function.php.

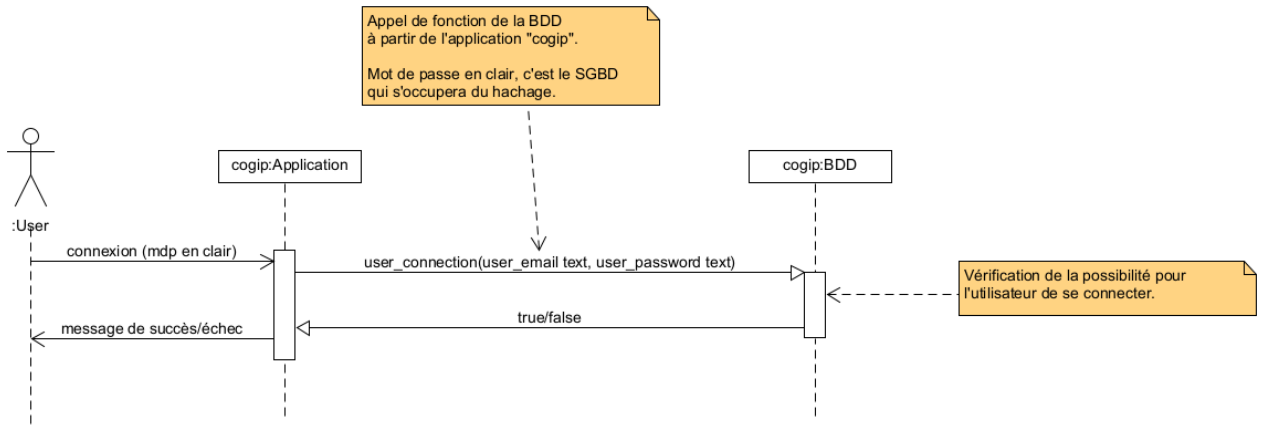
Pour la vérification de l'email il vous est conseillé d'utiliser une regex (vous pourrez vous inspirer de l'exemple suivant : https://stackoverflow.com/questions/10164758/postgresql-regex-to-validate-email-addresses).

2.13 FONCTION DE CONNEXION D'UN UTILISATEUR

2.13.1 Descriptif

L'objectif est de mettre en place une fonction permettant de gérer la connexion d'un utilisateur avec un accès sécurisé.

Voici un diagramme de séquence présentant le processus de connexion d'un utilisateur.



Vous allez gérer la partie connexion à la base de données en implémentant la fonction « **user_connection** » indiquée sur le diagramme de séquence.

Cette fonction prend deux paramètres :

- l'adresse email de l'utilisateur ;
- le mot de passe non haché de l'utilisateur.

La fonction retournera un booléen indiquant si la connexion de l'utilisateur est acceptée ou non.

Règle fondamentale : un mot de passe en base de données est stocké sous sa version « hachée », la fonction devra donc calculer la valeur de hachage du mot de passe fourni en paramètre.

Vous utiliserez la méthode « sha-1 » dans le cadre de ce développement (cf. TP de gestion des commandes).

2.13.2 Code à écrire

Etape 1 : création de la fonction

Utilisez le code suivant pour créer la fonction de connexion d'un utilisateur :

```
CREATE OR REPLACE function user_connection(user_email text, user_password text)
 RETURNS boolean
 LANGUAGE plpgsql
as  $function$
declare
    user_id_reference int; -- l'identifiant de l'utilisateur récupéré en base de données
    user_password_reference text; -- le mot de passe de l'utilisateur récupéré en base de données
    user_exists boolean; -- un indicateur d'existence de l'utilisateur
    hashed_password text; -- va contenir le mot de passe haché
begin

    -- vérification de l'existence de l'utilisateur
    user_exists = exists(select *
                         from "user" u
                         where u.email like user_email);

    -- si l'utilisateur existe, on vérifie son mot de passe
    if user_exists then
        -- récupération du mot de passe stocké en BDD
        select "password"
        into user_password_reference
        from "user" u
        where u.email like user_email;

        -- calcul du hash du mot de passe passé en paramètre et vérification avec le hash en BDD
        hashed_password = encode(digest(user_password, 'sha1'), 'hex');
        if hashed_password like user_password_reference then
            return true;
        end if;
    end if;

    -- alert pour l'utilisateur
    raise notice 'L''utilisateur ayant pour email % n''existe pas en base de données.',
user_email;
    return false;
END
$function$;
```

Etape 2 : ajout de la date de dernière connexion

Ajouter à votre fonction une requête permettant de mettre à jour la table « user » afin de stocker la date et l'heure de la connexion lorsque la connexion est acceptée.

Etape 3 : ajout d'une fonctionnalité de blocage du compte

A chaque tentative de connexion infructueuse en raison d'un mot de passe erroné il faudra incrémenter la valeur de « connexion_attempt ».

Dans le cas de 3 tentatives infructueuses le compte devra être bloqué (modification de la valeur de « blocked_account »).

3. CREATION DE DECLENCHEURS (TRIGGERS)

3.1 QU'EST-CE QU'UN DECLENCHEUR ?

Un déclencheur (ou « trigger ») est une procédure stockée qui se déclenche automatiquement juste **après ou avant** les requêtes suivantes :

- « INSERT »
- « UPDATE »
- « DELETE »

Les déclencheurs permettent **d'automatiser des actions** au sein d'un SGBD, on peut par exemple citer les opérations suivantes :

- **Journalisation** des requêtes sur une base de données (pour une surveillance de l'activité) ;
- Ajouter des règles de validation pour autoriser ou non les requêtes (particulièrement utile pour les requêtes « INSERT » ou « UPDATE »).

3.2 DECLENCHEURS AFFICHANT DES MESSAGES

Vous allez découvrir les différents types de déclencheurs pouvant être utilisés.

Dans un premier temps, vous mettrez en place plusieurs déclencheurs simples affichant des messages lorsque des requêtes de plusieurs types sont effectuées.

3.2.1 Message avant un « INSERT »

3.2.1.1 Descriptif de la fonction

Mettre en place un déclencheur qui surviendra **avant l'ajout** d'un enregistrement dans la table « supplier » et qui affiche « Un ajout de fournisseur va être fait. Le nouveau fournisseur est **<nom-fournisseur>**».

Pour créer un déclencheur en PostgreSQL, il vous faut procéder en deux temps :

1. Création d'une fonction/procédure qui contient le code PL/pgSQL qui doit être exécuté ;
2. Création du déclencheur qui appellera la fonction/procédure lors d'un évènement qu'on définira.

3.2.1.2 Création de la fonction

Créez la fonction d'affichage du message en utilisant le code suivant :

```
CREATE OR REPLACE FUNCTION display_message_on_supplier_insert()
 RETURNS trigger
 LANGUAGE plpgsql
AS $$
BEGIN
  raise notice '« Un ajout de fournisseur va être fait. Le nouveau fournisseur est %', NEW.name;
  return null;
```

```
END;
$$
```

Règles à suivre

Une fonction qui a pour objectif d'être déclenchée doit suivre ces règles :

- Elle ne doit **pas avoir de paramètres** ;
- Son type de retour doit être « **trigger** » ;
- Elle doit renvoyer **soit « null »**, **soit un enregistrement** ayant exactement la même structure que la table concernée par la fonction.

Variables utilisables

Dans les fonctions de déclencheurs, plusieurs **variables spéciales** sont utilisables. Parmi elles, on retrouve les variables « **NEW** » et « **OLD** ».

Ces variables sont de type « RECORD » (type définissant une ligne d'une table) et font référence aux lignes impactées par la requête à l'origine de l'évènement.

Ainsi, nous avons :

- **OLD :** la ligne **avant la modification** commandée par la requête (valable pour un « DELETE » et un « UPDATE »).
- **NEW :** la ligne **après modification** (valable pour un « INSERT » et « UPDATE »)

Il est possible de faire référence aux colonnes des enregistrements « OLD » et « NEW » en utilisant le point **(« . »)** et le nom de la colonne d'intérêt : « **NEW.name** » permet, par exemple, de retrouver le nouveau nom du fournisseur.

D'autres variables spéciales sont disponibles, vous pourrez trouverez des détails concernant ces variables ici : https://pgdocptbr.sourceforge.io/pg82/plpgsql-trigger.html

3.2.1.3 Création du déclencheur

L'action permettant de déclencher l'évènement sera un « INSERT », vous allez créer un déclencheur prenant en compte cet évènement et appelant la fonction précédemment créée.

Voici le code que vous pourrez utiliser :

```
create trigger before_insert_supplier -- "before_insert_supplier" est le nom du déclencheur
before insert -- indication sur le type d'évènement du déclencheur
on public.supplier -- nom de la table concernée
for each row -- quand se déclencher ? ROW ou statement (explication ci-dessous)
execute function display_message_on_supplier_insert(); -- appel de la fonction lorsque le
déclencheur s'active
```

`before insert` : indication sur le type d'évènement de déclenchement. Ici « avant insertion ». Vous pouvez utiliser « before » ou « after » ainsi que les indicateurs sur les requêtes SQL « insert », « update » et « delete » (ainsi que le radical ordre « truncate »).

`for each row` : deux options sont disponibles ici, « for each row » et « for each statement ». « for each row » indique que la fonction sera appelée à chaque fois qu'une ligne sera impactée par une requête. « for each statement » indique que la fonction est appelée à chaque requête (le requête peut avoir un impact sur plusieurs lignes).

3.2.2 Message après un « UPDATE »

Modifiez le code proposé précédemment pour afficher le message « Mise à jour de la table des fournisseurs. » dans le cas d'une mise à jour (ordre « UPDATE »).

La fonction affichera les informations de l'ancienne ligne ainsi que celles de la nouvelle (pensez aux variables « OLD » et « NEW).

3.3 DECLENCHEURS EMPECHANT UNE REQUETE

Il est possible d'**empêcher qu'une requête puisse se faire** en utilisant un déclencheur.

Pour se faire, il faut **lever une exception** dans la fonction déclenchée par un évènement.

3.3.1 Empêcher la suppression de l'administrateur principal

3.3.1.1 Descriptif

On souhaite empêcher la suppression de l'utilisateur ayant pour rôle « MAIN_ADMIN ».

Le code de la fonction permettant de le faire est le suivant :

```
CREATE OR REPLACE function check_user_delete()
 RETURNS trigger
 LANGUAGE plpgsql
AS $function$
begin
    if old.role = 'MAIN_ADMIN' then
        raise exception 'Impossible de supprimer l`''utilisateur %. Il s''agit de
l''administrateur principal.', old.id;
    end if;
  return null;
END;
$function$
```

11. Créez un déclencheur de type « before delete » appelant cette nouvelle fonction

3.3.2 Empêcher la suppression des commandes non livrées

3.3.2.1 Descriptif

La fonction « check_orderline_delete » a pour objectif d'empêcher la suppression des enregistrements de la table « order_line » pour les commandes qui n'ont pas été livrées complétement (« delivered_quantity » est inférieure à « ordered_quantity »).

12. **Implémentez une fonction ainsi que son déclencheur permettant d'empêcher ce type de suppression.**

3.4 DECLENCHEUR DE MODIFICATION DE CONTENU DE TABLES

3.4.1 Modification des articles à commander

3.4.1.1 Descriptif

L'objectif est ici de créer un déclencheur qui permettra de mettre à jour une nouvelle table nommée « **items_to_order** » permettant de sauvegarder les articles pour lesquels le stock a atteint sa valeur d'alerte.

Vous utiliserez un déclencheur permettant de sauvegarder les informations concernant l'état du stock dans une nouvelle table « items_to_order » (structure de la table détaillée plus bas).

Le déclencheur devra s'activer en « before update » sur la table « item ».

3.4.1.2 Code à écrire

Etape 1 : création de la table

Créez une nouvelle table ayant le schéma suivant :

`items_to_order(id, #item_id, date_update, quantity)`

Suivant le dictionnaire de données suivant :

Attribut	Description	Type
id		serial
item_id	Clef étrangère vers un enregistrement de la table « item ».	int
quantity	Quantité d'objet en question à commande.	int
date_update	Date de dernière mise à jour de la quantité à commander.	date

Etape 2 : création d'une fonction qui à jour la table

Ecrivez une fonction trigger contenant un « update »

Etape 3 : création du déclencheur

Créez un déclencheur « after update » appelant la nouvelle fonction.

Etape 4 : empêcher la modification si la valeur est trop faible

Il ne doit pas être possible d'avoir un stock négatif.

Comment, à l'aide d'un nouveau déclencheur, faire en sorte d'empêcher la mise à jour dans ce cas de figure ?

13. Ecrivez le code de ce nouveau déclencheur

3.4.2 Table d'audit

3.4.2.1 Descriptif

Vous allez créer une nouvelle table permettant de stocker un enregistrement dans une table d'audit à chaque fois qu'une requête « INSERT », « DELETE » ou « UPDATE » est effectuée sur la table « item ».

Cette nouvelle table sera appelée « item_audit ».

3.4.2.2 Code à écrire

L'exemple « **37-3. A *PL/pgSQL* Trigger Procedure For Auditing »** de l'article situé à l'adresse https://pgdocptbr.sourceforge.io/pg82/plpgsql-trigger.html présente une proposition d'implémentation.

14. Inspirez-vous du code fourni pour créer votre table et développer le déclencheur approprié.

CREDITS

ŒUVRE COLLECTIVE DE l'AFPA
Sous le pilotage de la DIIP et du centre d'ingénierie sectoriel Tertiaire-Services

Equipe de conception (IF, formateur, mediatiseur)
Michel Coulard – Formateur Evry
Chantal Perrachon – IF Neuilly sur Marne

Date de mise à jour : 06/04/2022