IPEFA Sup Seraing - Verviers

Projet de développement SGDB

Notes de cours

Georgette Collard

2025-2026

Table des matières

Chapitre 10: PL/pgSQL

Tout d'abord PostgreSQL ne fait pas de différence entre une fonction et une procédure car tout est fonction dans PostgreSQL, même, dans une certaine mesure, pour les déclencheurs.

PL/pgSQL, signifie « *Programming Language / postgreSQL* » et sert justement à coder les routines (UDF, procédures et trigger) dans le langage interne à PostGreSQL au même titre que Transact SQL sert à MS SQL Server et PL/SQL à Oracle…

PL/pgSQL est un langage structuré en blocs. Le texte complet de la définition d'une fonction doit être un *bloc*. Un bloc est défini comme

```
[ <<label>> ]
[ DECLARE
    déclarations ]
BEGIN
    instructions
END [ label ];
```

Exemple :

```
CREATE FUNCTION une_fonction() RETURNS integer AS $$
DECLARE
    quantité integer := 30;
BEGIN
    RAISE NOTICE 'quantité vaut ici %', quantité;  -- quantité vaut ici 30
    quantité := 50;
    --
    -- Crée un sous-bloc
    --
    DECLARE
        quantité integer := 80;
    BEGIN
        RAISE NOTICE 'quantité vaut ici %', quantité;  -- quantité vaut ici 80
        /* commentaire
        sur plusieurs
        lignes */
    END;

    RAISE NOTICE 'quantité vaut ici %', quantité;  -- quantité vaut ici 50

    RETURN quantité;
END;
$$ LANGUAGE plpgsql;
```

Exécution :

```
select * from une_fonction();
```

Data Output | Messa

	une_fonction integer
1	50

Chaque déclaration et chaque expression au sein du bloc est terminé par un point-virgule. Un bloc qui apparaît à l'intérieur d'un autre bloc doit avoir un point-virgule après **END ;** néanmoins, le **END** final qui conclut le corps d'une fonction n'a pas besoin de point-virgule.

Tous les mots clés et identifiants peuvent être écrits en majuscules et minuscules mélangées. Les identifiants sont implicitement convertis en minuscule à moins d'être entourés de guillemets doubles

Commentaire

Double tiret (**- -**) : une ligne de commentaire

/* … ***/** : bloc de commentaire

Déclaration de variable

Toutes les variables utilisées dans un bloc doivent être déclarées dans la section déclaration du bloc. La seule exception est que la variable de boucle d'une boucle **FOR** effectuant une itération sur des valeurs entières est automatiquement déclarée comme variable entière (type *integer*)

La clause **DEFAULT**, si indiquée, spécifie la valeur initiale assignée à la variable quand on entre dans le bloc. Si la clause **DEFAULT** n'est pas indiquée, la variable est initialisée à la valeur **SQL NULL**. L'option **CONSTANT** empêche l'assignation de la variable, de sorte que sa valeur reste constante pour la durée du bloc. Si **NOT NULL** est spécifié, l'assignement d'une valeur **NULL** aboutira à une erreur d'exécution.

Exemples :

```
quantité integer DEFAULT 32;
url varchar := 'http://mysite.com';
id_utilisateur CONSTANT integer := 10;
```

Paramètres de fonctions

```
CREATE FUNCTION prixTTC(real, real) returns real as $$
BEGIN
    RETURN $1 * (1 + $2);
END;
$$ LANGUAGE plpgsql;
```

Pour donner un nom au paramètre, il existe deux façons :

```
CREATE FUNCTION prixTTC_nomParam(prix real, tva real) returns real as $$
BEGIN
    RETURN prix * (1 + tva);
END;
$$ LANGUAGE plpgsql;

CREATE FUNCTION prixTTC_alias(real, real) returns real as $$
DECLARE
    prix ALIAS FOR $1;
    tva ALIAS FOR $2;
BEGIN
    RETURN prix * (1 + tva);
END;
$$ LANGUAGE plpgsql;
```

Fonction déclarée avec des paramètres de sortie

```
CREATE FUNCTION prixTTC_sortie(prix real, tva real, out ttc real) as $$
BEGIN
    ttc =  prix * (1 + tva);
END;
$$ LANGUAGE plpgsql;
```

Les paramètres en sortie sont utiles lors du retour de plusieurs valeurs.

```
CREATE FUNCTION somme_n_produits(x int, y int, OUT somme int, OUT produit int) AS $$
BEGIN
    somme := x + y;
    produit := x * y;
END;
$$ LANGUAGE plpgsql;
```

L'exemple ci-dessous a le même résultat :

On crée un type composite anonyme pour le résultat de la fonction.

```
CREATE TYPE somme_produit AS (somme int, produit int);

CREATE FUNCTION somme_n_produits_bis(x int, y int) RETURNS somme_produit AS $$
BEGIN
    RETURN (x + y, x * y );
END;
$$ LANGUAGE plpgsql;
```

Copie de type

Exemple

```
id_utilisateur utilisateurs.id_utilisateur%TYPE;
```

On n'a pas besoin de connaître le type de données de *id_utilisateur*. Si le type de *id_utilisateur* change dans le futur (**integer** en real par exemple), on n'a pas besoin de changer la définition de fonction.

%TYPE est particulièrement utile dans le cas de fonctions polymorphes puisque les types de données nécessaires aux variables internes peuvent changer d'un appel à l'autre. Des variables appropriées peuvent être créées en appliquant **%TYPE** aux arguments de la fonction ou à la variable fictive de résultat.

Type ligne (%ROWTYPE)

Une variable de type composite est appelée variable *ligne* (ou variable *row-type*). Une telle variable peut contenir une ligne entière de résultat de requête **SELECT** ou **FOR**, du moment que l'ensemble de colonnes de la requête correspond au type déclaré de la variable. Les champs individuels de la valeur *row* sont accessibles en utilisant la notation pointée, par exemple *varligne.champ.*

```
create function detail_livre
    (liv_ligne LIVRE) returns text
as $$
declare
    cat_ligne CATEGORIE%rowtype;
begin
    select * into cat_ligne from categorie
    where (idCategorie = liv_ligne.idCategorie);
    return liv_ligne.isbn
    || ';' || liv_ligne.titre || ';'
        || liv_ligne.auteur || ';' || liv_ligne.edition || ';'
        || cat_ligne.nom  ;

end;
$$ LANGUAGE plpgsql;
```

```
select  detail_livre(l.*)
from livre l
where (isbn = '9782754750905');
```

Data Output | Notifications

	essai_detail_livre text
1	9782754750905;La nature par la plume;Stefan Dahlvist;Du Panthéon;Poesie;

Type record

Les variables record sont similaires aux variables de type ligne mais n'ont pas de structure prédéfinie. Elles empruntent la structure effective de type ligne de la ligne à laquelle elles sont affectées durant une commande **SELECT** ou **FOR**. La sous-structure d'une variable record peut changer à chaque fois qu'on l'affecte. Une conséquence de cela est qu'elle n'a pas de sous-structure jusqu'à ce qu'elle ait été affectée, et toutes les tentatives pour accéder à un de ses champs entraînent une erreur d'exécution.

La structure réelle de la ligne n'est pas connue quand la fonction est écrite mais, dans le cas d'une fonction renvoyant un type record, la structure réelle est déterminée quand la requête appelante est analysée, alors qu'une variable record peut changer sa structure de ligne à la volée.

Plusieurs fonctions avec le même nom - polymorphisme

```
CREATE OR REPLACE FUNCTION compute_due_date(DATE) RETURNS DATE AS $$
DECLARE
    due_date    DATE;
    rental_period   INTERVAL := '7 days';
BEGIN
    due_date := $1 + rental_period;
    RETURN due_date;
END;
$$ LANGUAGE plpgsql;

CREATE FUNCTION OR REPLACE compute_due_date(DATE, INTERVAL) RETURNS DATE AS $$
BEGIN
    RETURN ($1 + $2);
END;
$$ LANGUAGE plpgsql;

CREATE FUNCTION OR REPLACE compute_due_date(dans INTERVAL) RETURNS DATE AS $$
BEGIN
    RETURN current_date + dans;
END;
$$ LANGUAGE plpgsql;
```

```
select  compute_due_date( interval '1 MONTH') ;

select  compute_due_date(current_date, interval '1 MONTH') ;

select  compute_due_date(current_date) ;
```

Instruction de base

Assignation

identifiant := *expression*;

Exemples :

```
id_utilisateur := 20;
tax := sous_total * 0.06;
```

Instruction IF

```
IF condition THEN
   instructions
[ ELSIF condition THEN
   instructions ]
[ ELSIF condition THEN
   instructions ]
[ ELSE
   instructions ]
END IF;
```

Instruction CASE

```
CASE search-expression
   WHEN expression [, expression [...]] THEN
      instructions
   WHEN expression [, expression [...]] THEN
      instructions
   ...
   [ ELSE
      instructions ]
END CASE;
```

```
CASE x
    WHEN 1, 2 THEN
        nbre := nbre + 1;
    ELSE
        nbre := nbre - 1;
END CASE;
```

```
CASE
   WHEN bool-expression  THEN
      instructions
   WHEN bool-expression THEN
      instructions
   ...
   [ ELSE
      instructions ]
END CASE;
```

```
CASE
    WHEN w < 0 THEN
        signe := 'negatif';
    ELSE
        signe := 'positif';
END CASE;
```

```
CASE
   WHEN bool-expression  THEN
      expression
   WHEN bool-expression THEN
      expression
   ...
   [ ELSE
      expression ]
END CASE;
```

```
signe := CASE
    WHEN w < 0 THEN
        'negatif';
    ELSE
        'positif';
END CASE;
```

Boucle simple

Instruction loop

```
[<<label>>]
LOOP
   instructions
END LOOP [ label ];
```

Instruction exit

```
EXIT [ label ] [ WHEN expression ];
```

Si aucun *label* n'est donné, la boucle la plus imbriquée se termine et l'instruction suivant **END LOOP** est exécutée. Si un *label* est donné, ce doit être le label de la boucle, du bloc courant ou d'un niveau moins imbriqué. La boucle ou le bloc nommé se termine alors et le contrôle continue avec l'instruction située après le **END** de la boucle ou du bloc correspondant.

Si **WHEN** est spécifié, la sortie de boucle ne s'effectue que si *expression* vaut true. Sinon, le contrôle passe à l'instruction suivant le **EXIT**.

```
LOOP
     x:=x+1;
     IF x>10 THEN
          EXIT;
     END IF;
END LOOP;

LOOP
     x:=x+1;
     EXIT WHEN x>10;
END LOOP;

<<toto>>
LOOP
   x:=x+1;
   LOOP
     y:=y-1;
     EXIT toto WHEN y<x;
   END LOOP;
END LOOP;
```

Instruction continue

```
CONTINUE [ label ] [ WHEN expression ];
```

CONTINUE à l'intérieur de la boucle aura le même effet qu'en langage C, on retourne au début de la boucle (itération suivante).

Instruction while

```
[<<label>>]
WHILE expression LOOP
   instructions
END LOOP [ label ];
```

Instruction FOR

```
[<<label>>]
FOR iterator IN [ REVERSE ] expression .. expression [bY expr ] LOOP
   instructions
END LOOP [ label ];
```

La variable *iterator* est automatiquement déclarée comme une variable de type INTEGER. Dans la version avec une étiquette *iterator* peut-être préfixée par l'étiquette.

Exemples :

```
FOR I IN  1..10 LOOP
      --I Prendra les valeurs 1,2,...,10
END LOOP;

FOR I IN REVERSE 10..9 LOOP
      --I Prendra les valeurs 10,9,8,7,...,1

 FOR I IN REVERSE 10..1 BY 2 LOOP
     --I prendra les valeurs 10,8,6,4,2
 END LOOP;

 <<toto>>
   FOR i IN 1..n LOOP
     k:=k * toto.i;
   END LOOP;
```

SELECT … INTO et la variable FOUND

On peut récupérer le résultat d'une requête SELECT grâce au **SELECT … INTO …**

```
SELECT liste-select INTO destination FROM ...;
```

La requête **SELECT** ne doit pas retourner plus qu'une ligne ou une valeur.

La variable destination est une variable ligne ou une liste de variable simple séparé par des virgules.

FOUND permet de vérifier si la requête a effectivement retourné une ligne ou non.

FOUND est une variable booléenne affectée par un **SELECT INTO,** et peut être immédiatement testée après cette commande pour savoir si le select a retourné un résultat

```
CREATE FUNCTION isLivreEmprunte(exemplaire.idLivre%type) RETURNS text AS $$
DECLARE
    t_ligne Emprunt%ROWTYPE;    -- Déclaration d'une variable de type ligne
                                -- de la table EMPRUNT
BEGIN
    -- fait le select dans t_ligne
    SELECT * INTO t_ligne FROM EMPRUNT
        where (idLivre = $1) and (date_retour is null);
    IF FOUND THEN
        RETURN 'Ce livre est emprunté';       -- select a trouvé une ligne
    ELSE
        RETURN 'Ce livre n''est pas emprunté'; -- select n'a pas trouvé de ligne
    END IF;
END;
$$ LANGUAGE plpgsql;
```

Fonction retournant des tables

Il y a deux manière de déclarer qu'une fonction retourne une table, soit en utilisant **RETURNS SETOF** *sometype* soit avec **RETURNS TABLE**(*nom1 type1,...,nomn typen*). La table de *résulta*t est construite avec les appels à **RETURN QUERY** et/ou **RETURN NEXT**

Returns setof

Une fonction qui retourne une table déclare la valeur de retour comme **RETURNS SETOF** *sometype* Dans l'exemple suivant on suppose qu'il existe la table nommée *LIVRE*. Donc la fonction *listerLivre()* retourne une table dont chaque ligne a exactement le même type qu'une ligne de la table *LIVRE*.

```
CREATE FUNCTION listerLivre() RETURNS SETOF LIVRE AS $$
BEGIN
    -- le select suivant peut retourner plusieurs lignes.
    -- Sans SETOF le select retourne une seule lignes.
    RETURN QUERY SELECT * FROM LIVRE order by auteur, titre;
END;
$$ LANGUAGE plpgsql;
```

RETURNS TABLE

La fonction suivante retourne une table avec les attributs nomCategorie-souscatégorie (nom de la catégorie et la sous-catégorie concaténée) et isbn, titre, auteur des livres disponibles.

Ajouter des éléments dans la table de résultats : RETURN QUERY et RETURN NEXT

Exemple : Retourner une table avec les attributs attributs nomCategorie-souscatégorie (nom de la catégorie et la sous-catégorie concaténée) et isbn, titre, auteur des livres disponibles.

RETURN NEXT

```
CREATE FUNCTION CategorieLivre()
RETURNS TABLE(nom_sous_categorie varchar(80),
              isbn livre.isbn%type,
              titre livre.titre%type,
              auteur livre.auteur%type) AS $$

declare r record;

BEGIN
    for r in select  categorie.nom as nom, sousCategorie,
                     livre.isbn as isbn,  livre.titre as titre,
                     livre.auteur as auteur
              from  livre inner join categorie
                     on livre.idCategorie = categorie.idCategorie
               order by categorie, sousCategorie, auteur, titre
    loop

          nom_sous_categorie = r.nom || ' ' || r.sousCategorie;
          isbn = r.isbn;
          titre = r.titre;
          auteur = r.auteur;
          return next;          -- ceci ajoute (nom_sous_categorie, isbn, titre, auteur)
                                   -- dans le résultat

    end loop;
    return;
    END;
$$ LANGUAGE plpgsql;
```

RETURN QUERY

```
CREATE FUNCTION CategorieLivre_2()
RETURNS TABLE(nom_sous_categorie varchar(80),
              isbn livre.isbn%type,
              titre livre.titre%type,
              auteur livre.auteur%type) AS $$

declare r record;
declare nomPlusSousCategorie varchar(80);

BEGIN
    for r in select  categorie.nom as nom, sousCategorie,
                     livre.isbn as isbn,  livre.titre as titre,
                     livre.auteur as auteur
              from  livre inner join categorie
                     on livre.idCategorie = categorie.idCategorie
               order by categorie, sousCategorie, auteur, titre
    loop
         -- remarque :
         -- concaténation d'une chaine de caractères à une chaîne null
         -- donne comme résultat une chaîne null

           if (r.sousCategorie is null)
           then
               nomPlusSousCategorie = r.nom ;
            else

               nomPlusSousCategorie = r.nom || ' ' || r.sousCategorie;
            end if;

            -- ajoute un tuple dans la table résultat
            return query values(nomPlusSousCategorie , r.isbn, r.titre, r.auteur);

    end loop;
    return;
    END;
$$ LANGUAGE plpgsql;
```

On peut faire la même chose sans boucle :

```
CREATE or replace FUNCTION CategorieLivre_3()
RETURNS TABLE(nom_sous_categorie text,
              isbn livre.isbn%type,
              titre livre.titre%type,
              auteur livre.auteur%type) AS $$

declare r record;
declare nomPlusSousCategorie varchar(80);

BEGIN
    return query select categorie.nom || ' ' || sousCategorie,
                    livre.isbn ,  livre.titre ,
                    livre.auteur
                 from  livre inner join categorie
                    on livre.idCategorie = categorie.idCategorie
                 where sousCategorie is not null
                 order by categorie, sousCategorie, auteur, titre;

    return query select categorie.nom || ' ', livre.isbn ,  livre.titre ,
                    livre.auteur
                 from  livre inner join categorie
                    on livre.idCategorie = categorie.idCategorie
                 where sousCategorie is  null
                 order by categorie, sousCategorie, auteur, titre;

    return;
    END;
$$ LANGUAGE plpgsql;
```

Ces 3 fonctions *CategorieLivre, CategorieLivre _1, CategorieLivre_2* sont identiques.

Soit la fonction :

```
CREATE  function CategorieSousCategorie()
RETURNS TABLE(nom_sous_categorie text, idCategorie integer) AS $$

declare r record;

BEGIN
    for r in select * from CATEGORIE
                    order by nom, sousCategorie
    LOOP

        if (r.sousCategorie is null)
        then
            nom_sous_categorie := r.nom ;
        else
            nom_sous_categorie := r.nom || ' ' || r.sousCategorie;
        end if;
        idCategorie := r.idCategorie;
        return next;    -- ceci ajoute (nomPlusSousCategorie) dans le résultat
    end loop;
    return;
    END;
$$ LANGUAGE plpgsql;
```

Les 4 commandes suivantes donnent le même résultat :

```
select * from CategorieLivre();

select * from CategorieLivre_2();

select * from CategorieLivre_3();

select nom_sous_categorie, isbn, titre, auteur, edition
from CategorieSousCategorie() as t, livre
where t.idCategorie = livre.idCategorie;
```

Les 3 fonctions suivantes sont identiques

```
CREATE OR REPLACE FUNCTION InfoLivre(livre.isbn%TYPE)
    RETURNS TABLE (titre livre.titre%TYPE,
                   auteur livre.auteur%TYPE,
                   edition livre.edition%TYPE)  AS $$
DECLARE
    r record;

BEGIN
    FOR r IN SELECT livre.titre, livre.auteur, livre.edition
    FROM livre where isbn = $1
    LOOP
        RETURN QUERY VALUES (r.titre, r.auteur, r.edition);
    END LOOP;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE OR REPLACE FUNCTION InfoLivre1(livre.isbn%TYPE,
                                      OUT titre livre.titre%TYPE,
                                      OUT auteur livre.auteur%TYPE,
                                      OUT edition livre.edition%TYPE)  AS $$

BEGIN
    SELECT livre.titre, livre.auteur, livre.edition
    INTO titre, auteur, edition FROM livre where isbn = $1;
    RETURN ;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE OR REPLACE FUNCTION InfoLivre0(livre.isbn%TYPE)  returns livre AS $$

declare t_ligne LIVRE%rowtype;

BEGIN
    SELECT * INTO t_ligne FROM livre where isbn = $1;
    RETURN t_ligne;
END;
$$ LANGUAGE plpgsql;
```

```
select * from InfoLivre1('9782213630236');

select * from InfoLivre1('9782213630236');

select titre, auteur, edition from InfoLivre0('9782213630236');
```

Si ce livre existe, les 3 commandes ci-dessus donneront le même résultat

Data Output | Explain | Messages | Notifications

	titre character varying	auteur character varying	edition character varying
1	Toute la geographie ...	Jean-Claude Barreau	Fayard

Par contre, si le livre n'existe pas, *InfoLivre* ne retourne aucune ligne, par contre *InfoLivre0* et *InfoLivre1* retourne une ligne dont les valeurs des champs sont mis à null.

```
select * from InfoLivre('9882213630236');
```

Data Output | Explain | Messages | Notifications

	titre character varying	auteur character varying	edition character varying

```
select * from InfoLivre1('9882213630236');
```

Data Output | Explain | Messages | Notifications

	titre character varying	auteur character varying	edition character varying
1	[null]	[null]	[null]

```
select titre, auteur, edition from InfoLivre0('9882213630236');
```

Data Output | Explain | Messages | Notifications

	titre character varying (100)	auteur character varying (50)	edition character varying (30)
1	[null]	[null]	[null]

Gestion des erreurs

Le gestionnaire d'exception (bloc de code **EXCEPTION**) permet de récupérer la main sur le code après une erreur.

La fonction **RAISE** permet de lancer une erreur ou un avertissement

bloc de gestion des erreurs

Le bloc de code **EXCEPTION** doit être situé en fin de corps avant le mot clé **END**.

```
EXCEPTION
  WHEN <erreur11>  [ OR <erreur12>  [ OR ... ] ]
  THEN
     ...
[ WHEN <erreur21>  [ OR <erreur22>  [ OR ... ] ]
  THEN
     ...  ]
[ ... ]
```

Où

```
<erreurN> ::=
  { erreur_SQL | SQLSTATE 'valeur' | OTHERS }
```

erreur_SQL : est le code alphabétique d'une erreur
SQLSTATE 'valeur' : permet d'indiquer une classe d'erreur
OTHERS : permet de trapper toutes les erreurs.

Exemple

```
CREATE FUNCTION F_NOM_JOUR(d DATE)
RETURNS VARCHAR(8)
AS
$code$
DECLARE retval VARCHAR(8);
BEGIN
   CASE EXTRACT(DOW FROM d)
      WHEN 1 THEN retval := 'Lundi';
      WHEN 2 THEN retval := 'Mardi';
      WHEN 3 THEN retval := 'Mercredi';
      WHEN 4 THEN retval := 'Jeudi';
      WHEN 5 THEN retval := 'Vendredi';
      WHEN 6 THEN retval := 'Samedi';
      WHEN 7 THEN retval := 'Dimanche';
   END CASE;
   RETURN retval;
EXCEPTION
   WHEN case_not_found
   THEN RETURN 'inconnu';
END;
$code$
LANGUAGE PLPGSQL
```

Exécution :

```
SELECT F_NOM_JOUR('17/11/2019') ;
```

	f_nom_jour character varying
1	inconnu

Le 17/11/2019 est un dimanche, dans ce cas *d* vaut 0 et ne se trouve pas dans la définition du **case**.

GET DIAGNOSTICS

GET DIAGNOSTICS nbre = **ROW_COUNT** ;

nbre contiendra le nombre de lignes traitées par la dernière commande SQL exécutée.

Exemple

Cette fonction retourne le nombre de lignes insérées.

```
CREATE  FUNCTION ajoutCategorie(varchar(30), varchar(50))
RETURNS integer as $$
DECLARE
    nb_insert integer;
BEGIN
    INSERT INTO CATEGORIE(nom, sousCategorie) values ($1, $2);
    GET DIAGNOSTICS nb_insert = ROW_COUNT;
    return nb_insert;
END;
$$ LANGUAGE plpgsql;
```

Récupérer l'identifiant après une insertion

Cette fonction retourne le nombre de ligne insérée et l'identifiant de la nouvelle personne.

```
CREATE  or replace FUNCTION ajoutPersonne(personne.nom%TYPE,
                            personne.prenom%TYPE,
                             personne.gsm%type,
                            adresse.rue%TYPE,
                            adresse.cp%TYPE,
                            adresse.localite%TYPE,
                             out nb_insert integer,
                             out idpersonne personne.idPersonne%TYPE)
 as $$

BEGIN
    INSERT INTO PERSONNE ( nom, prenom, gsm, adressePersonne)
    VALUES ($1, $2, $3, ($4, $5, $6))
    returning personne.idPersonne into idPersonne;
    GET DIAGNOSTICS nb_insert = ROW_COUNT;

END;
$$ LANGUAGE plpgsql;
```

Génération d'une erreur

Il est possible de générer une erreur à l'aide de la commande **RAISE** dont la syntaxe est la suivante :

```
RAISE [ { DEBUG | LOG | INFO | NOTICE | WARNING | EXCEPTION } ]
    [ { 'message_erreur' [ <expressions> ] | erreur_SQL | SQLSTATE 'valeur' } ]
[ USING { MESSAGE | DETAIL | HINT | ERRCODE } = 'message' ];
```

DEBUG, LOG, INFO : enregistre l'erreur dans le journal PostGreSQL. Le code n'est pas affecté et aucun gestionnaire d'erreur (bloc **EXCEPTION**) ne la prend en compte.

NOTICE, WARNING : enregistre l'erreur dans le journal PostGreSQL et l'envoie au client à titre d'avertissement. Le code n'est pas affecté et aucun gestionnaire d'erreur (bloc **EXCEPTION**) ne la prend en compte.

EXCEPTION : enregistre l'erreur dans le journal PostGreSQL, l'envoie au client à titre d'erreur, annule automatiquement la transaction, interrompt le code et passe le contrôle au gestionnaire d'erreur (bloc **EXCEPTION**) s'il y en a un.

EXCEPTION est la valeur par défaut dans la liste facultative suivant le mot clé **RAISE**.

La chaîne de caractères *message_erreur* peut contenir des caractères **%** qui seront remplacé par les *expressions* à la position ordinale considérée.

Les journaux PostGreSQL sont situé dans l'arborescence d'installation du serveur dans *\PostgreSQL\[N° de version)\data\pg_log*

```
CREATE  or replace FUNCTION F_CONSTRUCT_DATE(an INTEGER, mois INTEGER, jour INTEGER)
RETURNS DATE
AS $$

DECLARE ds VARCHAR(10);
        d  DATE;

BEGIN
   -- test des cnditions limites
   IF an <= 0
   THEN
     RAISE EXCEPTION 'Année % en dehors de la plage de valeur 1..9999', an;
   END IF;
   IF mois NOT BETWEEN 1 AND 12
   THEN
     RAISE EXCEPTION 'Mois % en dehors de la plage de valeur 1..12', mois
          USING HINT = 'Mois incorrect, mais année ' || CAST(an AS VARCHAR(4)) || ' correcte.';
   END IF;
   IF jour NOT BETWEEN 1 AND 31
   THEN
     RAISE EXCEPTION 'Jour % en dehors de la plage de valeur 1..31', jour
          USING HINT = 'Mois incorrect, mais année ' || CAST(an   AS VARCHAR(4)) || ' et mois '
                                                     || CAST(mois AS VARCHAR(4)) || ' correct.';
   END IF;
   IF jour = 31 AND mois IN (2, 4, 6, 9, 11)
   THEN
     RAISE EXCEPTION 'Jour 31 impossible pour le mois %', mois
          USING HINT = 'Mois ou jour incorrect, mais année ' || CAST(an AS VARCHAR(4)) || ' correcte.';
   END IF;
   IF jour = 30 AND mois = 2
   THEN
     RAISE EXCEPTION 'Jour 30 impossible pour le mois de février'
          USING HINT = 'Rectifiez les paramètres de date.';
   END IF;

   IF jour = 29 AND mois = 2 and (an % 4) <> 0  and ((an % 100) = 0 or ((an % 100) = 0 and (an % 400) <> 0))
   THEN
     RAISE EXCEPTION 'Jour 29 impossible pour le mois de février de l''année %', an
          USING HINT = 'Rectifiez les paramètres de date. Les années bissextiles sont divisibles par 4.';
   END IF;

   -- nos paramètres sont bons, construction de la date
   ds := CAST(an AS VARCHAR(4));
   WHILE CHARACTER_LENGTH(ds) < 4
   LOOP
      ds := '0' || ds;
   END LOOP;
   IF mois < 10
   THEN
     ds := ds || '-0' || CAST(mois AS VARCHAR(1));
   ELSE
     ds := ds || '-'  || CAST(mois AS VARCHAR(2));
   END IF;
   IF jour < 10
   THEN
     ds := ds || '-0' || CAST(jour AS VARCHAR(1));
   ELSE
     ds := ds || '-'  || CAST(jour AS VARCHAR(2));
   END IF;
   d := CAST(ds AS DATE);
   RETURN d;
END;
$$
LANGUAGE PLPGSQL
```

Déclencheur

Un déclencheur spécifie que la base de données doit exécuter automatiquement une fonction donnée chaque fois qu'un certain type d'opération est exécuté. Les fonctions déclencheur peuvent être attachées à une table, une vue.

La fonction déclencheur doit être définie avant que le déclencheur lui-même puisse être créé. La fonction déclencheur doit être déclarée comme une fonction ne prenant aucun argument et retournant un type trigger (la fonction déclencheur reçoit ses entrées via une structure TriggerData passée spécifiquement, et non pas sous la forme d'arguments ordinaires de fonctions).

Une fois qu'une fonction déclencheur est créée, le déclencheur (trigger) est créé avec **CREATE TRIGGER.** Une même fonction déclencheur peut être utilisable par plusieurs déclencheurs.

Voici la syntaxe de la création d'un déclencheur

```
CREATE TRIGGER trigger_name
    {BEFORE | AFTER } {event}
    ON table_name
    [FOR [EACH] {ROW | STATEMENT } ]
        EXECUTE PROCEDURE trigger_function;
```

1) D'abord, on spécifie le nom du déclencheur après les mots clés **CREATE TRIGGER**.
2) Ensuite, on spécifie le moment que le déclencheur se déclenche. Cela peut être **BEFORE** ou **AFTER** avant ou après qu'un événement se produise.
3) Ensuite, on spécifie l'événement qui appelle le déclencheur. L'événement peut être **INSERT, DELETE, UPDATE**
4) Ensuite, on spécifie le nom de la table associé au déclencheur après le mot-clé **ON**.
5) Ensuite, on spécifie le type de déclencheurs qui peut être :

   - Déclencheur au niveau de la ligne qui est spécifié par la clause **FOR EACH ROW**.
   - Déclencheur au niveau de l'instruction qui est spécifié par la clause **FOR EACH STATEMENT**.

Quand une fonction PL/pgSQL est appelée en tant que trigger, plusieurs variables spéciales sont créées automatiquement dans le bloc de plus haut niveau. Comme par exemple :

NEW

Type de données RECORD ; variable contenant la nouvelle ligne de base de données pour les opérations **INSERT** / **UPDATE** dans les triggers de niveau ligne. Cette variable est non initialisée dans un trigger de niveau instruction et pour les opérations **DELETE**.

OLD

Type de données RECORD ; variable contenant l'ancienne ligne de base de données pour les opérations **UPDATE**/**DELETE** dans les triggers de niveau ligne. Cette variable est non initialisée dans les triggers de niveau instruction et pour les opérations **INSERT**.

Exemples

Déclencheur de vérification

Création de la fonction déclencheur

```
CREATE FUNCTION trigger_personne() RETURNS trigger
LANGUAGE PLPGSQL
as $$
BEGIN
    -- Vérifie que le nom est donné et contient au moins deux caractères
    IF NEW.nom is NULL or char_length(new.nom) < 5 THEN
        RAISE EXCEPTION
            '% n''est pas un nom valide, le nom doit contenir au moins 5 caractères',
            NEW.nom;
    END IF;
    IF NEW.prenom is NULL or char_length(new.prenom) < 5 THEN
        RAISE EXCEPTION
            '% n''est pas un prénom valide, le prénom doit contenir au moins 5 caractères',
            NEW.prenom;
    END IF;

    RETURN NEW;
END;

$$
```

Création du déclencheur

```
CREATE TRIGGER newOrModPersonne BEFORE INSERT OR UPDATE ON personne
    FOR EACH ROW EXECUTE PROCEDURE trigger_personne();
```

Exemple 2

Supposons que si le nom change ou le prénom change, on veut avoir une trace dans une table personne_audit

```
CREATE TABLE personne_audits
(
    id serial primary key,
    personne_id int not null,
    last_nom varchar(15) not null,
    last_prenom varchar(15) not null,
    changed_on TIMESTAMP(6) not null
);
```

Création de la fonction déclencheur

```
CREATE OR REPLACE FUNCTION log_last_nom_prenom_changes()
  RETURNS TRIGGER
  LANGUAGE PLPGSQL
  AS
$$
BEGIN
    IF NEW.nom <> OLD.nom or  NEW.prenom <> OLD.prenom  THEN
        INSERT INTO personne_audits
            (personne_id,last_nom, last_prenom, changed_on)
        VALUES(OLD.idpersonne,OLD.nom, OLD.prenom,now());
    END IF;

    RETURN NEW;
END;
$$
```

Création du déclencheur

```
CREATE TRIGGER last_nom_prenom_changes
  BEFORE UPDATE
  ON personne
  FOR EACH ROW
  EXECUTE PROCEDURE log_last_nom_prenom_changes();
```

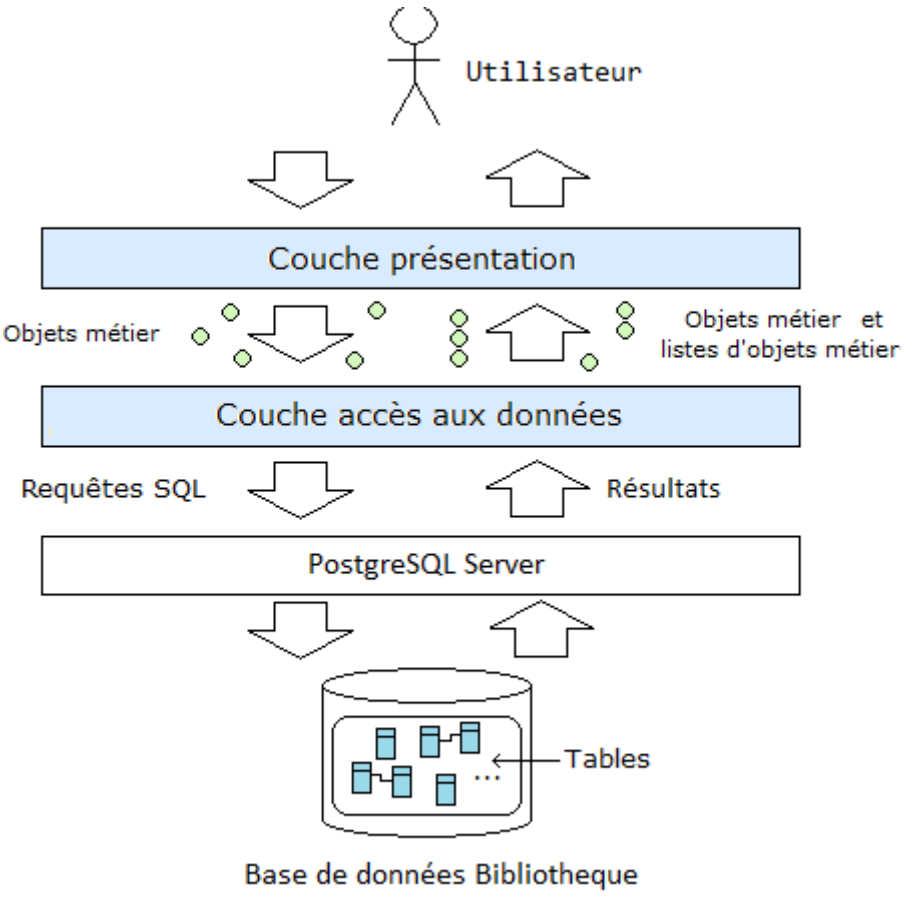
Chapitre 11 : Création des procédures stockées

Reprenons le modèle d'application en 2 couches

1) Couche présentation (*presentation layer*): cette couche gère les accès à la console (clavier et écran) et effectue les traitements sur les données : tests, calculs, ... Dans notre application, il va s'agir d'une classe appelée **Presentation**.

2) Couche accès aux données (*data access layer*): cette couche inclut toute classe d'accès à la base de données. Dans notre application, il va s'agir de la classe **AccesBD**.

Voici une figure montrant ou se situe chaque couche:

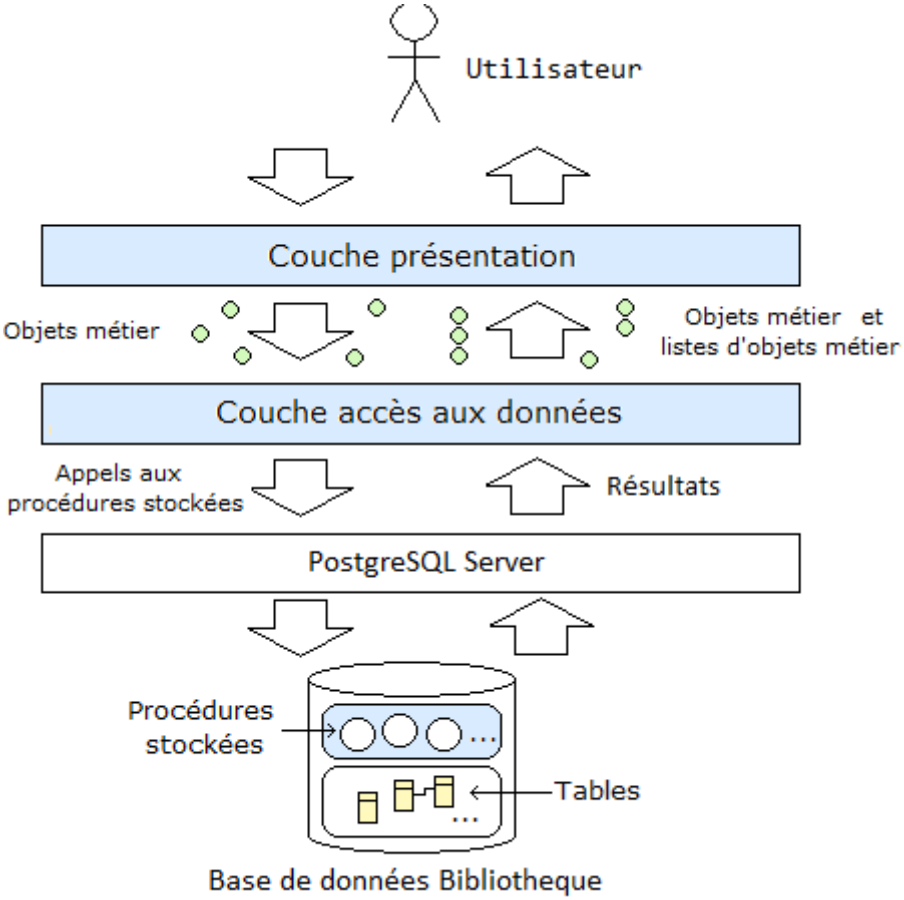


Pendant l'exécution de l'application, les objets des classes *Presentation* et *AccesBD* s'échangent des objets de classes directement construites à partir des tables de la base de données (Livre, Personne, ...). Ces classes sont appelées **classes métier**.

Grace à ce modèle en 2 couches, nous allons pouvoir modifier la couche accès aux données effectuant les accès à la base de données sans avoir à ajouter, modifier ou supprimer une seule ligne de code dans les autres classes.

Concrètement, dans cette nouvelle version de l'application *Bibliotheque*, nous allons transférer toutes les requêtes SQL depuis les méthodes de la classe *AccesBD* vers des **procédures stockées**.

Voici a quoi maintenant ressemble l'application:



Dans cette architecture, la couche accès aux données a été divisée en 2 parties:

1) La classe *AccesBD* a pour rôle d'appeler les procédures stockées qui sont mémorisées dans la base de données *Bibliotheque*.

2) Les procédures stockées accèdent aux tables de la base de données.

Rien ne va changer dans l'application par rapport à la version précédente, a l'exception du contenu du fichier *AccesBD.cs* se trouvant dans l'espace de noms *coucheAccesBD*.

On va ajouter les procédures stockées suivantes dans la base de données :

```
CREATE FUNCTION listerCategorie() RETURNS SETOF CATEGORIE AS $$
BEGIN
    -- le select suivant peut retourner plusieurs lignes.
    -- Sans SETOF le select retourne une seule lignes.
    RETURN QUERY SELECT * FROM CATEGORIE order by categorie, sousCategorie;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION listerLivreCategorie(categorie.idCategorie%type)
RETURNS SETOF LIVRE AS $$
BEGIN
    -- le select suivant peut retourner plusieurs lignes.
    -- Sans SETOF le select retourne une seule lignes.
    RETURN QUERY SELECT * FROM LIVRE where idCategorie = $1;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE OR REPLACE FUNCTION InfoLivre(livre.isbn%TYPE)
    RETURNS TABLE (titre livre.titre%TYPE,
                   auteur livre.auteur%TYPE,
                   edition livre.edition%TYPE)  AS $$
DECLARE
    r record;

BEGIN
    FOR r IN SELECT livre.titre, livre.auteur, livre.edition
    FROM livre where isbn = $1
    LOOP
        RETURN QUERY VALUES (r.titre, r.auteur, r.edition);
    END LOOP;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE or replace FUNCTION listerLivreDisponible(livre.titre%type, livre.auteur%type)
returns table (idlivre exemplaire.idlivre%type, isbn exemplaire.isbn%type)
AS $$
declare r record;
BEGIN
    for r in select  LIVRE.isbn as isbn, exemplaire.idLivre as idlivre
            from EXEMPLAIRE inner join LIVRE on EXEMPLAIRE.isbn = LIVRE.isbn
            where exemplaire.idLivre not in  (select EXEMPLAIRE.idLivre
                                from (EXEMPLAIRE inner join EMPRUNT
                                 on EXEMPLAIRE.idLivre = EMPRUNT.idLivre)
                                 inner join LIVRE on livre.isbn = exemplaire.isbn
                                where (date_retour is null))
                                       and titre = $1 and auteur = $2
        loop
            return query values (r.idlivre, r.isbn);
        end loop;
END;
$$ LANGUAGE plpgsql;
```

```sql
CREATE or replace FUNCTION listerLivreEmprunte(livre.titre%type, livre.auteur%type)
returns table (idlivre exemplaire.idlivre%type, isbn exemplaire.isbn%type)
AS $$
declare r record;
BEGIN

    for r in select livre.isbn as isbn, EMPRUNT.idLivre as idLivre
             from (EXEMPLAIRE inner join EMPRUNT
                   on EXEMPLAIRE.idLivre = EMPRUNT.idLivre)
                    inner join LIVRE on livre.isbn = exemplaire.isbn
             where date_retour is null and titre = $1 and auteur = $2
        loop
            return query values (r.idlivre, r.isbn);
        end loop;

END;
$$ LANGUAGE plpgsql;
```

```sql
CREATE FUNCTION listerPersonne() RETURNS SETOF PERSONNE AS $$
BEGIN
    -- le select suivant peut retourner plusieurs lignes.
    -- Sans SETOF le select retourne une seule lignes.
    RETURN QUERY SELECT * FROM PERSONNE order by nom, prenom;
END;
$$ LANGUAGE plpgsql;
```

```sql
CREATE FUNCTION infoPersonne(personne.idPersonne%type)
RETURNS SETOF PERSONNE AS $$
BEGIN
    -- le select suivant peut retourner plusieurs lignes.
    -- Sans SETOF le select retourne une seule lignes.
    RETURN QUERY SELECT * FROM PERSONNE where idPersonne = $1;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE  FUNCTION ajoutLivre(categorie.idCategorie%TYPE,
                                    livre.titre%TYPE,
                                    livre.auteur%type,
                                    livre.edition%TYPE,
                                    livre.isbn%TYPE,
                                    out nb_insert integer,
                                    out idLivre exemplaire.idLivre%TYPE)
 as $$

BEGIN


        INSERT INTO LIVRE ( idCategorie, titre, auteur, edition, isbn)
        VALUES ($1, $2, $3, $4, $5) ;
        GET DIAGNOSTICS nb_insert = ROW_COUNT;
        INSERT INTO EXEMPLAIRE (  isbn)
        VALUES ($5)
        returning EXEMPLAIRE.idLivre into idLivre;
        GET DIAGNOSTICS nb_insert = ROW_COUNT;


END;
$$ LANGUAGE plpgsql;
```

```
CREATE  FUNCTION ajoutExemplaire(livre.isbn%TYPE,
                            out nb_insert integer,
                            out idLivre exemplaire.idLivre%TYPE)
 as $$

BEGIN
    INSERT INTO EXEMPLAIRE ( isbn)
    VALUES ($1)
    returning EXEMPLAIRE.idLivre into idLivre;
    GET DIAGNOSTICS nb_insert = ROW_COUNT;

END;
$$ LANGUAGE plpgsql;
```

```
CREATE or replace function SupprimeLivre(exemplaire.idLivre%TYPE)

RETURNS integer AS $$
DECLARE
    nb_delete integer DEFAULT 0;
    isbnExemplaire livre.isbn%type;
    nbre integer;
BEGIN
    select exemplaire.isbn from exemplaire where idLivre = $1 into isbnExemplaire;
    select count(*) from exemplaire where exemplaire.isbn = isbnExemplaire into nbre;
    delete from exemplaire where idLivre=$1;
    get diagnostics nb_delete = ROW_count;
    if nb_delete = 1 and nbre =1 then
       nb_delete = 0;
       delete from livre where livre.isbn = isbnExemplaire;
       get diagnostics nb_delete = ROW_count;
        if nb_delete = 0 then
            ROLLBACK;
        end if;
    END IF;
    RETURN nb_delete;

END;
$$ LANGUAGE plpgsql;
```

```
CREATE  FUNCTION ajoutpersonne(personne.nom%TYPE,
                               personne.prenom%TYPE,
                               personne.gsm%TYPE,
                               adresse.rue%TYPE,
                               adresse.cp%TYPE,
                               adresse.localite%TYPE)
 RETURNS integer
as $$
DECLARE
    nb_insert integer;
    idPersonne integer default 0;
BEGIN
    INSERT INTO PERSONNE( nom, prenom, gsm, adressepersonne)
    VALUES ($1, $2, $3, ($4, $5, $6))
    returning PERSONNE.idpersonne into idPersonne;

    return idPersonne;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE  FUNCTION ajoutEmprunt(personne.idPersonne%TYPE,
                                    exemplaire.idLivre%TYPE,
                                    out nb_insert integer)
 as $$

BEGIN
    INSERT INTO EMPRUNT ( idPersonne, idLivre, date_sortie)
    VALUES ($1, $2, current_date) ;
    GET DIAGNOSTICS nb_insert = ROW_COUNT;

END;
$$ LANGUAGE plpgsql;
```

```
CREATE  FUNCTION prolongerEmprunt(personne.idPersonne%TYPE,
                                    exemplaire.idLivre%TYPE,
                                    out nb_update integer)
 as $$

BEGIN
    update EMPRUNT
    set date_prolongation = current_date
    where idPersonne =$1 and idLivre = $2;
    GET DIAGNOSTICS nb_update = ROW_COUNT;

END;
$$ LANGUAGE plpgsql;
```

```
CREATE  FUNCTION restituerEmprunt(personne.idPersonne%TYPE,
                                    exemplaire.idLivre%TYPE,
                                    out nb_update integer)
 as $$

BEGIN
    update EMPRUNT
    set date_retour = current_date
    where idPersonne =$1 and idLivre = $2;
    GET DIAGNOSTICS nb_update = ROW_COUNT;

END;
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION ExisteCategorie(categorie.nom%type, categorie.souscategorie%type)
RETURNS integer AS $$
DECLARE
    t_ligne categorie%ROWTYPE;
BEGIN
    SELECT * INTO t_ligne FROM categorie
    WHERE (nom = $1 and sousCategorie = $2);

    IF FOUND THEN
        RETURN 1;
    ELSE
        RETURN 0;
    END IF;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION ExisteLivre(exemplaire.idlivre%type)
RETURNS integer AS $$
-- returns 0 si exemplaire n'existe pas
-- returns 1 si l'exemplaire n'est pas emprunté
-- returns 2 si l'exemplaire est emprunté
DECLARE
    trouve integer default 0;
BEGIN
    SELECT * FROM EXEMPLAIRE
    WHERE (idLivre = $1);

    IF FOUND THEN
        SELECT * from EMPRUNT
        where idLivre = $1 and date_retour is null ;
        IF FOUND THEN
            trouve:= 2;
        ELSE
            trouve := 1;
        END IF;
    END IF;
    return trouve;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION ExisteEmprunt(personne.idPersonne%type, exemplaire.idlivre%type)
RETURNS integer AS $$
DECLARE
    trouve integer;
BEGIN
    SELECT case when date_prolongation is null
                            then 2
                            else 1 end
    INTO trouve FROM Emprunt
    WHERE (idPersonne = $1 and idLivre = $2
        and date_retour is null);

    IF FOUND THEN
        RETURN trouve;
    ELSE
        RETURN 0;
    END IF;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION ExisteLivre(exemplaire.idlivre%type)
RETURNS integer AS $$
-- returns 0 si exemplaire n'existe pas
-- returns 1 si l'exemplaire n'est pas emprunté
-- returns 2 si l'exemplaire est emprunté
DECLARE
    exemplaire_ligne exemplaire%ROWTYPE;
    emprunt_ligne   emprunt%ROWTYPE;
BEGIN
    SELECT * FROM EXEMPLAIRE into exemplaire_ligne
    WHERE (idLivre = $1);

    IF FOUND THEN
        SELECT * from EMPRUNT into emprunt_ligne
        where idLivre = $1 and date_retour is null ;
        IF FOUND THEN
            RETURN 2;
        ELSE
            RETURN 1;
        END IF;
    ELSE
        RETURN 0;
    END IF;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION ExistePersonne(personne.idPersonne%type)
RETURNS integer AS $$
DECLARE
    t_ligne personne%ROWTYPE;
BEGIN
    SELECT * INTO t_ligne FROM personne
    WHERE idPersonne = $1;

    IF FOUND THEN
        RETURN 1;
    ELSE
        RETURN 0;
    END IF;
END;
$$ LANGUAGE plpgsql;
```

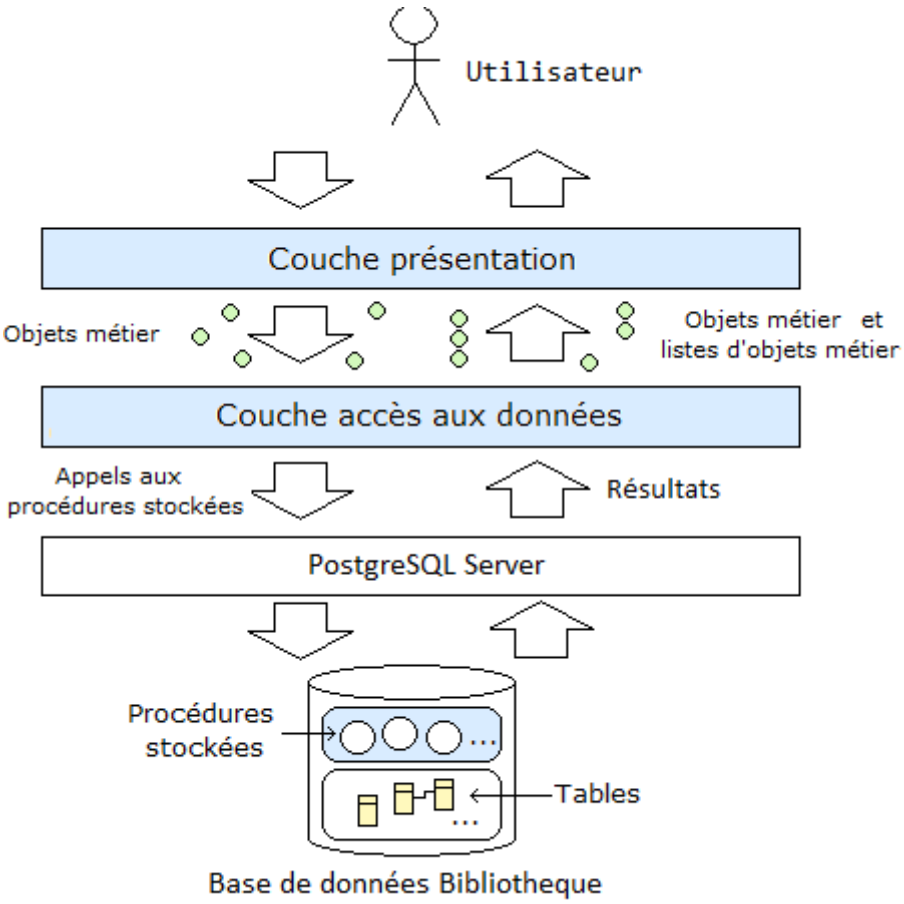
```
CREATE  FUNCTION getLivreISBN(livre.isbn%TYPE)
    RETURNS TABLE (titre livre.titre%TYPE,
                   auteur livre.auteur%TYPE,
                   edition livre.edition%TYPE,
                   idCategorie categorie.idCategorie%type)
AS $$
DECLARE
    r record;

BEGIN
    FOR r IN SELECT livre.titre, livre.auteur,
                    livre.edition, livre.idCategorie
    FROM livre where isbn = $1
    LOOP
        RETURN QUERY VALUES (r.titre, r.auteur, r.edition, r.idCategorie);
    END LOOP;
END;
$$ LANGUAGE plpgsql;
```

Chapitre 12 : modification de la couche accès aux données

Nous avons vu que la couche accès aux données de l'application est à présent constituée de 2 parties: la classe *AccesBD* et les procédures stockées.

- La classe *AccesBD* a pour rôle d'appeler les procédures stockées qui sont mémorisées dans la base de données *Bibliotheque*.

- Les procédures stockées accèdent aux tables de la base de données.



Classe AccesBD

Vous ne trouverez plus aucune requête SQL dans les méthodes dans la classe *AccesBD*, car les requêtes SQL se trouvent dans les procédures stockées :

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Npgsql;
using Bibliotheque.classeMetier;

namespace Bibliotheque.coucheAccesBD
{
    class AccesBD
    {
        private NpgsqlConnection SqlConn;
        /**
        * Constructeur : créer une instance de la classe AccesBD
*/
        public AccesBD()
        {
            try
            {

                this.SqlConn = new NpgsqlConnection("Server=localhost;" +
                "port=5432;" +
                "Database=Bibliotheque;" +
                "UserID=postgres;" +
                "Password = postgreSQL");


                this.SqlConn.Open();
            }
            catch (Exception e)
            {
                throw new ExceptionAccesBD("Connexion à la BD", e.Message);
            }
        }
```

```
// Obtenir la liste des catégories

public List<Categorie> ListeCategorie()
{
    List<Categorie> liste = null;
    NpgsqlCommand sqlCmd = null;

    try
    {
        sqlCmd = new NpgsqlCommand("select idCategorie, nom, " +
                                   "sousCategorie from listerCategorie()", this.SqlConn);

        NpgsqlDataReader sqlreader = sqlCmd.ExecuteReader();

        if (sqlreader.Read())
        {
            liste = new List<Categorie>();

            do
            {
                liste.Add(new Categorie(Convert.ToInt32(sqlreader["idCategorie"]),
                                        Convert.ToString(sqlreader["nom"]),
                                        Convert.ToString(sqlreader["sousCategorie"])));
            } while (sqlreader.Read());

        }
        sqlreader.Close();
    }

    catch (Exception e)
    {
        throw new ExceptionAccesBD(sqlCmd.CommandText, e.Message);
    }

    return liste;
}
```

```
//Obtenir la liste des livres appartenant à une catégorie
//
// entrée : idCategorie (catégorie.idCategorie)
// sortie la liste des exemplaires ordonné par auteur et titre

public List<Livre> ListeLivreCategorie(Categorie categorie)
{
    List<Livre> liste = null;
    NpgsqlCommand sqlCmd = null;

    try
    {
        sqlCmd = new NpgsqlCommand("select  isbn, titre, auteur, " +
                                   " edition from ListerLivreCategorie(:idCategorie) ",

                                   this.SqlConn);

        // Ajoute les paramètres

        sqlCmd.Parameters.Add(new NpgsqlParameter("idCategorie",
            NpgsqlTypes.NpgsqlDbType.Integer));


        // Prepare la commande

        sqlCmd.Prepare();

        // Ajouter les valeurs aux paramètres

        sqlCmd.Parameters[0].Value = categorie.Idcategorie;

        NpgsqlDataReader sqlreader = sqlCmd.ExecuteReader();

        if (sqlreader.Read())
        {

            liste = new List<Livre>();

            do
            {
                liste.Add(new Livre(categorie.Idcategorie,
                    Convert.ToString(sqlreader["isbn"]),
                    Convert.ToString(sqlreader["titre"]),
                    Convert.ToString(sqlreader["auteur"]),
                    Convert.ToString(sqlreader["edition"])));
            } while (sqlreader.Read());

        }
        sqlreader.Close();


        return liste;
    }

    catch (Exception e)
    {

        throw new ExceptionAccesBD(sqlCmd.CommandText, e.Message);
    }

}
```

```
// Obtenir les informations d'une personne
//
// entrée: identifiant de la personne
// sortie: entité de la classe personne

public Personne InfoPersonne(int idpersonne)
{
    NpgsqlCommand sqlCmd = null;
    Personne personne = null;

    try
    {
        sqlCmd = new NpgsqlCommand("select nom, prenom, GSM, " +
                                   "(adressePersonne).rue as rue, " +
                                   "(adressePersonne).localite as localite, " +
                                   "(adressePersonne).cp as cp " +
                                   " from InfoPERSONNE (:idPersonne)",
                                   this.SqlConn);

        // Ajoute les paramètres

        sqlCmd.Parameters.Add(new NpgsqlParameter("idPersonne",
            NpgsqlTypes.NpgsqlDbType.Integer));


        // Prepare la commande

        sqlCmd.Prepare();

        // Ajouter les valeurs aux paramètres

        sqlCmd.Parameters[0].Value = idpersonne;

        NpgsqlDataReader sqlreader = sqlCmd.ExecuteReader();


        if (sqlreader.Read())
        {
            personne = new Personne(idpersonne,
                Convert.ToString(sqlreader["nom"]),
                Convert.ToString(sqlreader["prenom"]),
                Convert.ToString(sqlreader["gsm"]),
                new Adresse(Convert.ToString(sqlreader["rue"]),
                Convert.ToString(sqlreader["cp"]),
                Convert.ToString(sqlreader["localite"])));
        }

        sqlreader.Close();
        return personne;

    }

    catch (Exception e)
    {
        throw new ExceptionAccesBD(sqlCmd.CommandText, e.Message);
    }
}
```

```
// Ajout d'une catégorie

public int AjouterCategorie(Categorie categorie)
{
    NpgsqlCommand sqlCmd = null;
    try
    {
        sqlCmd = new NpgsqlCommand("select * from ajoutCategorie " +
        "( :nom, :sousCategorie)", this.SqlConn);

        // Ajoute les paramètres

        sqlCmd.Parameters.Add(new NpgsqlParameter("nom",
            NpgsqlTypes.NpgsqlDbType.Varchar));
        sqlCmd.Parameters.Add(new NpgsqlParameter("sousCategorie",
            NpgsqlTypes.NpgsqlDbType.Varchar));

        // Prepare la commande

        sqlCmd.Prepare();

        // Ajouter les valeurs aux paramètres

        sqlCmd.Parameters[0].Value = categorie.Nom;
        sqlCmd.Parameters[1].Value = categorie.Souscategorie;

        // Execute la commande

        return (sqlCmd.ExecuteNonQuery());
    }
    catch (Exception e)
    {
        throw new ExceptionAccesBD(sqlCmd.CommandText, e.Message);
    }
}
```

```
    public Livre GetLivreISBN(string isbn)
    {
        NpgsqlCommand sqlCmd = null;
        Livre livre = null;

        try
        {
            sqlCmd = new NpgsqlCommand("select auteur, titre, edition, idCategorie " +
                                       " from getLivreISBN( :isbn )",
                                      this.SqlConn);

            // Ajoute les paramètres

            sqlCmd.Parameters.Add(new NpgsqlParameter("isbn",
                NpgsqlTypes.NpgsqlDbType.Varchar));


            // Prepare la commande

            sqlCmd.Prepare();

            // Ajouter les valeurs aux paramètres

            sqlCmd.Parameters[0].Value = isbn;


            NpgsqlDataReader sqlreader = sqlCmd.ExecuteReader();

        if (sqlreader.Read())
        {
            livre = new Livre();
            livre.Isbn = isbn;
            livre.Auteur = Convert.ToString(sqlreader["auteur"]);
            livre.Titre = Convert.ToString(sqlreader["titre"]);
            livre.Edition = Convert.ToString(sqlreader["edition"]);

            livre.Idcategorie =
                Convert.ToInt32(sqlreader["idCategorie"]);

        }

        sqlreader.Close();
        return livre;
    }
    catch (Exception e)
    {
        throw new ExceptionAccesBD(sqlCmd.CommandText, e.Message);
    }
}
```

```
// Ajouter un livre

public int AjouterLivre(Livre livre)
{
    NpgsqlCommand sqlCmd = null;
    try
    {
        sqlCmd = new NpgsqlCommand("select * from ajoutLivre" +
            //sqlCmd = new NpgsqlCommand("select * from testTransaction" +
            "(:idCategorie, :titre, :auteur,  " +
            ":edition, :isbn)", this.SqlConn);

        // Ajoute les paramètres

        sqlCmd.Parameters.Add(new NpgsqlParameter("idCategorie",
            NpgsqlTypes.NpgsqlDbType.Integer));
        sqlCmd.Parameters.Add(new NpgsqlParameter("titre",
            NpgsqlTypes.NpgsqlDbType.Varchar));
        sqlCmd.Parameters.Add(new NpgsqlParameter("auteur",
            NpgsqlTypes.NpgsqlDbType.Varchar));
        sqlCmd.Parameters.Add(new NpgsqlParameter("edition",
            NpgsqlTypes.NpgsqlDbType.Varchar));
        sqlCmd.Parameters.Add(new NpgsqlParameter("isbn",
            NpgsqlTypes.NpgsqlDbType.Varchar));

        // Prepare la commande

        sqlCmd.Prepare();

        // Ajouter les valeurs aux paramètres

        sqlCmd.Parameters[0].Value = livre.Idcategorie;
        sqlCmd.Parameters[1].Value = livre.Titre;
        sqlCmd.Parameters[2].Value = livre.Auteur;
        sqlCmd.Parameters[3].Value = livre.Edition;
        sqlCmd.Parameters[4].Value = livre.Isbn;


        // Execute la commande

        return (sqlCmd.ExecuteNonQuery());
    }
    catch (Exception e)
    {
        throw new ExceptionAccesBD(sqlCmd.CommandText, e.Message);
    }
}
```

```
// Ajouter un exemplaire

public int AjouterExemplaire(Livre livre)
{
    NpgsqlCommand sqlCmd = null;
    try
    {
        sqlCmd = new NpgsqlCommand("select * from ajoutExemplaire(:isbn)",
            this.SqlConn);

        // Ajoute les paramètres

        sqlCmd.Parameters.Add(new NpgsqlParameter("isbn",
            NpgsqlTypes.NpgsqlDbType.Varchar));


        // Prepare la commande

        sqlCmd.Prepare();

        // Ajouter les valeurs aux paramètres

        sqlCmd.Parameters[0].Value = livre.Isbn;

        // Execute la commande

        return (sqlCmd.ExecuteNonQuery());
    }
    catch (Exception e)
    {
        throw new ExceptionAccesBD(sqlCmd.CommandText, e.Message);
    }
}
```

```
// Supprimer un exemplaire,
// si dernier exemplaire, le supprimer également dans la table LIVRE


// Supprimer un exemplaire,
// si dernier exemplaire, le supprimer également dans la table LIVRE

public int SupprimerLivre(Exemplaire exemplaire)
{

    NpgsqlCommand sqlCmd = null;
    try
    {
        sqlCmd = new NpgsqlCommand(
            "select * from supprimeLivre(:idLivre) ",
            this.SqlConn);

        // Ajout du paramètre

        sqlCmd.Parameters.Add(new NpgsqlParameter("idLivre",
            NpgsqlTypes.NpgsqlDbType.Integer));

        // Prepare la commande

        sqlCmd.Prepare();

        // Ajouter les valeurs aux paramètres

        sqlCmd.Parameters[0].Value = exemplaire.IdLivre;

        NpgsqlDataReader sqlreader = sqlCmd.ExecuteReader();

            if (sqlreader.Read())
            {
                int i = Convert.ToInt32(sqlreader[0]);
                sqlreader.Close();
                return i;
            }
            else
            {
                sqlreader.Close();
                return 0;
            }

        }

        catch (Exception e)
        {

            throw new ExceptionAccesBD(sqlCmd.CommandText, e.Message);
        }
    }


            // Execute la commande

            return (sqlCmd.ExecuteNonQuery());
        }
        catch (Exception e)
        {
            throw new ExceptionAccesBD(sqlCmd.CommandText, e.Message);
        }
    }
```

```
// Ajouter une Personne

public int AjouterPersonne(Personne personne)
{
    NpgsqlCommand sqlCmd = null;
    try
    {
        sqlCmd = new NpgsqlCommand("select * from ajoutPersonne" +
            "( :nom, :prenom, :gsm, :rue, :cp, :localite)",
            this.SqlConn);

        // Ajoute les paramètres

        sqlCmd.Parameters.Add(new NpgsqlParameter("nom",
            NpgsqlTypes.NpgsqlDbType.Varchar));
        sqlCmd.Parameters.Add(new NpgsqlParameter("prenom",
            NpgsqlTypes.NpgsqlDbType.Varchar));
        sqlCmd.Parameters.Add(new NpgsqlParameter("gsm",
            NpgsqlTypes.NpgsqlDbType.Varchar));
        sqlCmd.Parameters.Add(new NpgsqlParameter("rue",
            NpgsqlTypes.NpgsqlDbType.Varchar));
        sqlCmd.Parameters.Add(new NpgsqlParameter("cp",
            NpgsqlTypes.NpgsqlDbType.Varchar));
        sqlCmd.Parameters.Add(new NpgsqlParameter("localite",
            NpgsqlTypes.NpgsqlDbType.Varchar));

        // Prepare la commande

        sqlCmd.Prepare();

        // Ajouter les valeurs aux paramètres

        sqlCmd.Parameters[0].Value = personne.Nom;
        sqlCmd.Parameters[1].Value = personne.Prenom;
        sqlCmd.Parameters[2].Value = personne.GSM;
        sqlCmd.Parameters[3].Value = personne.adresse.rue;
        sqlCmd.Parameters[4].Value = personne.adresse.cp;
        sqlCmd.Parameters[5].Value = personne.adresse.localite;

        // Execute la commande

        return (sqlCmd.ExecuteNonQuery());
    }
    catch (Exception e)
    {
        throw new ExceptionAccesBD(sqlCmd.CommandText, e.Message);
    }
}
```

```
// Faire en sorte qu'un livre soit emprunt par une personne

public int AjouterEmprunt(Emprunt emprunt)
        {
            NpgsqlCommand sqlCmd = null;
            try
            {
                sqlCmd = new NpgsqlCommand("select * from ajoutEmprunt" +
                    "( :idLivre, :idPersonne) ",
                     this.SqlConn);

                // Ajoute les paramètres

                sqlCmd.Parameters.Add(new NpgsqlParameter("idLivre",
                    NpgsqlTypes.NpgsqlDbType.Integer));
                sqlCmd.Parameters.Add(new NpgsqlParameter("idPersonne",
                    NpgsqlTypes.NpgsqlDbType.Integer));

                // Prepare la commande

                sqlCmd.Prepare();

                // Ajouter les valeurs aux paramètres

                sqlCmd.Parameters[0].Value = emprunt.livreEmprunte.IdLivre;
                sqlCmd.Parameters[1].Value = emprunt.emprunteur.IdPersonne;

                // Execute la commande

                return (sqlCmd.ExecuteNonQuery());
            }
            catch (Exception e)
            {
                throw new ExceptionAccesBD(sqlCmd.CommandText, e.Message);
            }
        }
```

```
//Restituer un livre

public int RestituerLivre(Emprunt emprunt)
{
    NpgsqlCommand sqlCmd = null;
    try
    {
        sqlCmd = new NpgsqlCommand(
            "select * from restituerEmprunt(:idPersonne, :idLivre)",
            this.SqlConn);

        // Ajoute les paramètres

        sqlCmd.Parameters.Add(new NpgsqlParameter("idPersonne",
            NpgsqlTypes.NpgsqlDbType.Integer));
        sqlCmd.Parameters.Add(new NpgsqlParameter("idLivre",
           NpgsqlTypes.NpgsqlDbType.Integer));

        // Prepare la commande

        sqlCmd.Prepare();

        // Ajouter les valeurs aux paramètres

        sqlCmd.Parameters[0].Value = emprunt.emprunteur.IdPersonne;
        sqlCmd.Parameters[1].Value = emprunt.livreEmprunte.IdLivre;

        // Execute la commande

        return (sqlCmd.ExecuteNonQuery());
    }
    catch (Exception e)
    {
        throw new ExceptionAccesBD(sqlCmd.CommandText, e.Message);
    }
}
```

```
// Prolonger un emprunt

public int ProlongerEmprunt(Emprunt emprunt)
{
    NpgsqlCommand sqlCmd = null;
    try
    {
        sqlCmd = new NpgsqlCommand(
            "select * from prolongerEmprunt(:idPersonne, :idLivre)",
            this.SqlConn);

        // Ajoute les paramètres

        sqlCmd.Parameters.Add(new NpgsqlParameter("idPersonne",
            NpgsqlTypes.NpgsqlDbType.Integer));
        sqlCmd.Parameters.Add(new NpgsqlParameter("idLivre",
            NpgsqlTypes.NpgsqlDbType.Integer));

        // Prepare la commande

        sqlCmd.Prepare();

        // Ajouter les valeurs aux paramètres

        sqlCmd.Parameters[0].Value = emprunt.emprunteur.IdPersonne;
        sqlCmd.Parameters[1].Value = emprunt.livreEmprunte.IdLivre;

        // Execute la commande

        return (sqlCmd.ExecuteNonQuery());
    }
    catch (Exception e)
    {
        throw new ExceptionAccesBD(sqlCmd.CommandText, e.Message);
    }
}
```

```
// Obtenir la liste des livres (exemplaires) non empruntés pour un titre donné et
// un auteur donné

public List<Exemplaire> ListeLivreDisponible(string titre, string auteur)
{
    List<Exemplaire> liste = null;
    NpgsqlCommand sqlCmd = null;

    try
    {

        sqlCmd = new NpgsqlCommand("select idLivre, isbn " +
            " from listerLivreDisponible(:titre, :auteur)",

                                this.SqlConn);

        // Ajoute les paramètres

        sqlCmd.Parameters.Add(new NpgsqlParameter("titre",
            NpgsqlTypes.NpgsqlDbType.Varchar));

        sqlCmd.Parameters.Add(new NpgsqlParameter("auteur",
            NpgsqlTypes.NpgsqlDbType.Varchar));
        // Prepare la commande

        sqlCmd.Prepare();

        // Ajouter les valeurs aux paramètres

        sqlCmd.Parameters[0].Value = titre;
        sqlCmd.Parameters[1].Value = auteur;

        NpgsqlDataReader sqlreader = sqlCmd.ExecuteReader();
```

```
        if (sqlreader.Read())
        {
            liste = new List<Exemplaire>();

            do
            {
                liste.Add(new Exemplaire(Convert.ToString(sqlreader["isbn"]),
                    Convert.ToInt32(sqlreader["idLivre"])));
            } while (sqlreader.Read());
        }

        sqlreader.Close();

        return liste;
    }

    catch (Exception e)
    {
        throw new ExceptionAccesBD(sqlCmd.CommandText, e.Message);
    }
}
```

```
// Obtenir la liste des livres empruntés (exemplaires) pour un titre donné et
// un auteur donné

public List<Exemplaire> ListeLivreEmprunte(string titre, string auteur)
{
    List<Exemplaire> liste = null;
    NpgsqlCommand sqlCmd = null;

    try
    {

        sqlCmd = new NpgsqlCommand("select  isbn,  idLivre  " +
                                   "from listerLivreEmprunte(:titre, :auteur)",

                                   this.SqlConn);

        // Ajoute les paramètres

        sqlCmd.Parameters.Add(new NpgsqlParameter("titre",
            NpgsqlTypes.NpgsqlDbType.Varchar));

        sqlCmd.Parameters.Add(new NpgsqlParameter("auteur",
            NpgsqlTypes.NpgsqlDbType.Varchar));
        // Prepare la commande

        sqlCmd.Prepare();

        // Ajouter les valeurs aux paramètres

        sqlCmd.Parameters[0].Value = titre;
        sqlCmd.Parameters[1].Value = auteur;

        NpgsqlDataReader sqlreader = sqlCmd.ExecuteReader();
```

```
        if (sqlreader.Read())
        {
            liste = new List<Exemplaire>();

            do
            {
                liste.Add(new Exemplaire(Convert.ToString(sqlreader["isbn"]),
                    Convert.ToInt32(sqlreader["idLivre"])));
            } while (sqlreader.Read());
        }

        sqlreader.Close();

        return liste;
    }

    catch (Exception e)
    {
        throw new ExceptionAccesBD(sqlCmd.CommandText, e.Message);
    }
}
```

```
// Obtenir la liste des personnes

public List<Personne> ListePersonne()
{
    List<Personne> liste = null;
    NpgsqlCommand sqlCmd = null;

    try
    {
        sqlCmd = new NpgsqlCommand("select idPersonne, nom, " +
                                   "prenom, gsm," +
                                   " (adressePersonne).rue as rue, " +
                                    " (adressePersonne).cp as cp, " +
                                     " (adressePersonne).localite as localite " +
                                     " from listerPersonne()", this.SqlConn);

        NpgsqlDataReader sqlreader = sqlCmd.ExecuteReader();

        if (sqlreader.Read())
        {
            liste = new List<Personne>();

            do
            {
                liste.Add(new Personne(
                          Convert.ToInt32(sqlreader["idPersonne"]),
                          Convert.ToString(sqlreader["nom"]),
                          Convert.ToString(sqlreader["prenom"]),
                          Convert.ToString(sqlreader["gsm"]),
                          new Adresse(Convert.ToString(sqlreader["rue"]),
                             Convert.ToString(sqlreader["cp"]),
                             Convert.ToString(sqlreader["localite"]))));
            } while (sqlreader.Read());
        }


        sqlreader.Close();
    }

    catch (Exception e)
    {
        throw new ExceptionAccesBD(sqlCmd.CommandText, e.Message);
    }

    return liste;
}
```

```
// Verifie si une catégorie existe
//
// entrée : le nom de la catégorie et la sous-catégorie
//
// retourne 1 si la catégorie existe sinon retourne 0
public int ExisteCategorie(string nom, string souscategorie)
{
    NpgsqlCommand sqlCmd = null;
    int trouve = 0;
    try
    {
        sqlCmd = new NpgsqlCommand(
            "select * from ExisteCATEGORIE (:nom, :souscategorie)",
            this.SqlConn);

        // Ajoute les paramètres

        sqlCmd.Parameters.Add(new NpgsqlParameter("nom",
            NpgsqlTypes.NpgsqlDbType.Varchar));

        sqlCmd.Parameters.Add(new NpgsqlParameter("souscategorie",
            NpgsqlTypes.NpgsqlDbType.Varchar));

        // Prepare la commande

        sqlCmd.Prepare();

        // Ajouter les valeurs aux paramètres

        sqlCmd.Parameters[0].Value = nom;
        sqlCmd.Parameters[1].Value = souscategorie;

        NpgsqlDataReader sqlreader = sqlCmd.ExecuteReader();


        if (sqlreader.Read())
        {
            trouve = Convert.ToInt32(sqlreader[0]);
        }

        sqlreader.Close();

        return (trouve);
    }

    catch (Exception e)
    {

        throw new ExceptionAccesBD(sqlCmd.CommandText, e.Message);
    }
}
```

```
// Verifie si une personne existe
//
// entrée : l'identifiant de la personne
//
// retourne 1 si la personne existe sinon retourne 0
public int ExistePersonne(int idPersonne)
{
    NpgsqlCommand sqlCmd = null;
    int trouve = 0;
    try
    {
        sqlCmd = new NpgsqlCommand(
            "select * from existePersonne(:idPersonne)",
            this.SqlConn);

        // Ajoute les paramètres

        sqlCmd.Parameters.Add(new NpgsqlParameter("idPersonne",
            NpgsqlTypes.NpgsqlDbType.Integer));

        // Prepare la commande

        sqlCmd.Prepare();

        // Ajouter les valeurs aux paramètres

        sqlCmd.Parameters[0].Value = idPersonne;

        NpgsqlDataReader sqlreader = sqlCmd.ExecuteReader();

        if (sqlreader.Read())
        {
            trouve = Convert.ToInt32(sqlreader[0]);
        }

        sqlreader.Close();


        return (trouve);
    }

    catch (Exception e)
    {

        throw new ExceptionAccesBD(sqlCmd.CommandText, e.Message);
    }
}
```

```
// Vérifie si un emprunt existe
//
// entrée : identifiant de la personne
//          identifiant du livre
//
// //
// retourne 1 si ce livre est emprunté par cette personne
//            et peut être prolongé
// retourne 2 si ce livre est emprunté par cette personne
//            et a déjà été prolongé
// sinon retourne 0, si ce livre n'a pas été prolongé par cette personne


public int ExisteEmprunt(int idPersonne, int idLivre)
{
    NpgsqlCommand sqlCmd = null;
    int trouve = 0;
    try
    {
        sqlCmd = new NpgsqlCommand(
            "select  * from ExisteEmprunt(:idPersonne, :idLivre) ",
            this.SqlConn);

        // Ajoute les paramètres

        sqlCmd.Parameters.Add(new NpgsqlParameter("idLivre",
            NpgsqlTypes.NpgsqlDbType.Integer));
        sqlCmd.Parameters.Add(new NpgsqlParameter("idPersonne",
            NpgsqlTypes.NpgsqlDbType.Integer));

        // Prepare la commande

        sqlCmd.Prepare();


        // Ajouter les valeurs aux paramètres

        sqlCmd.Parameters[0].Value = idLivre;
        sqlCmd.Parameters[1].Value = idPersonne;

        NpgsqlDataReader sqlreader = sqlCmd.ExecuteReader();

        if (sqlreader.Read())
        {
            trouve = Convert.ToInt32(sqlreader[0]);
        }

        sqlreader.Close();

        return (trouve);
    }

    catch (Exception e)
    {
        throw new ExceptionAccesBD(sqlCmd.CommandText, e.Message);
    }
}
```

```
        public int ExisteLivre(int idLivre)
        {
            NpgsqlCommand sqlCmd = null;
            int trouve = 0;
            try
            {
                sqlCmd = new NpgsqlCommand(
                    "select * from existeLivre( :idLivre) ",
                    this.SqlConn);

                // Ajoute les paramètres

                sqlCmd.Parameters.Add(new NpgsqlParameter("idLivre",
                    NpgsqlTypes.NpgsqlDbType.Integer));


                // Prepare la commande

                sqlCmd.Prepare();

                // Ajouter les valeurs aux paramètres

                sqlCmd.Parameters[0].Value = idLivre;

                NpgsqlDataReader sqlreader = sqlCmd.ExecuteReader();

                if (sqlreader.Read())
                {
                    trouve = Convert.ToInt32(sqlreader[0]);
                }
                sqlreader.Close();

                return (trouve);
            }

            catch (Exception e)
            {
                throw new ExceptionAccesBD(sqlCmd.CommandText, e.Message);
            }
        }
    }
}
```