Automne 2016

Introduction à MYSQL

Saliha Yacoub
COLLEGE LIONEL-GROULX

Table des matières

Introduction 2
Les types de données : 2
Les entiers 2
NUMERIC et DECIMAL 3
FLOAT, DOUBLE et REAL 3
Les Dates 3
Pour les chaines de caractères : 4
Le type ENUM 4
Et les procédures et fonctions? 5
Et l'optimisation de requête ? 11
Rôle du système de privilèges 13
Quand les modifications de privilèges prennent-ils effets ? 14
Les commandes GRANTet REVOKE (à partir de la version 3.22.11 de MySQL) 14
Tableau des Privillèges pour GRANT and REVOKE 15
Exemples: 16
Application : Exercice ..... **Erreur ! Signet non défini.**
PDO (PHP Data Object) par l'exemple 18
Étapes pour le traitement des commandes SQL: 18
Étape 2 : envoyer des requêtes à la base de données : 20
Cas1 : Requêtes exécutées une seule fois 20
Cas 2 : Requêtes avec paramètres : 23
Cas3 : Appel de procédures. 28
Exemple 1 28
Exemple 2 30
Exemple 3 31
Compléments 32
PDOStatement::fetch 32
Description ¶ 32
Liste de paramètres ¶ 32
La classe PDOStatement 33
La classe PDO 34
Sources 36

Introduction

MySQL est le SGBDR libre et gratuit. MySQL est Open Source. Open Source (Standard Ouvert) signifie qu'il est possible à chacun d'utiliser et de modifier le logiciel. Tout le monde peut le télécharger sur Internet et l'utiliser sans payer aucun droit.

En général le SQL de MySQL est très semblable au SQL d'Oracle que nous avons vu les dernières sessions. Dans ce présent document, on ne présentera pas le SQL déjà connu. Seules les différences entre les deux SGBDs seront détaillées.

Vous pouvez télécharger le MySQL installer à
http://dev.mysql.com/downloads/mysql/#downloads

Même si ce SGBD est gratuit, comme il appartient à Oracle, vous avez besoin de créer votre compte Oracle. Mais si vous avez déjà ce compte, il vous suffit de vous loger.

Installation :
Après l'installation de MySQL, un compte root est créé. Ce compte possède tous les droits. Il est donc important de créer des comptes avec accès restreint afin de garantir la sécurité des données. Vous avez déjà créé ce compte.

Les types de données :

http://dev.mysql.com/doc/refman/5.7/en/date-and-time-types.html

Les entiers

Type	Storage (Bytes)	Minimum Value (Signed/Unsigned)	Maximum Value (Signed/Unsigned)
TINYINT	1	-128	127
		0	255
SMALLINT	2	-32768	32767
		0	65535
MEDIUMINT	3	-8388608	8388607
		0	16777215
INT	4	-2147483648	2147483647
		0	4294967295
BIGINT	8	-9223372036854775808	9223372036854775807

Type	Storage	Minimum Value	Maximum Value
	(Bytes)	(Signed/Unsigned)	(Signed/Unsigned)
		0	18446744073709551615

Si vous voulez que les nombre soit positif (non signés) utilisé Unsigned
Exemple :

```
create table eleves  (num int  UNSIGNED, nom varchar(2));
```

NUMERIC et DECIMAL

NUMERIC et DECIMAL sont équivalents et acceptent deux paramètres : la précision et l'échelle.

- La précision définit le nombre de chiffres significatifs stockés, donc les 0 à gauche ne comptent pas. En effet 0024 est équivalent à 24. Il n'y a donc que deux chiffres significatifs dans 0024.
- L'échelle définit le nombre de chiffres après la virgule.

Dans un champ DECIMAL(5,3), on peut donc stocker des nombres de 5 chiffres significatifs maximum, dont 3 chiffres sont après la virgule. Par exemple : 12.354, -54.258, 89.2 ou -56.
DECIMAL(4) équivaut à écrire DECIMAL(4, 0).(Comme le NUMBER de Oracle)

FLOAT, DOUBLE et REAL

Le mot-clé FLOAT peut s'utiliser sans paramètre, auquel cas quatre octets sont utilisés pour stocker les valeurs de la colonne. Il est cependant possible de spécifier une précision et une échelle, de la même manière que pour DECIMAL et NUMERIC.

Quant à REAL et DOUBLE, ils ne supportent pas de paramètres. DOUBLE est normalement plus précis que REAL (stockage dans 8 octets contre stockage dans 4 octets), mais ce n'est pas le cas avec MySQL qui utilise 8 octets dans les deux cas. Je vous conseille donc d'utiliser DOUBLE pour éviter les surprises en cas de changement de SGBDR.

Les Dates

Vous pouvez spécifier les valeurs des colonnes DATETIME, DATE

Le type DATE est prévu lorsque vous souhaitez stocker une date. MySQL affiche les valeurs de type DATE au format ‘AAAA-MM-JJ’. L'intervalle de validité va de '1000-01-01' à '9999-12-31'.

Le type DATETIME est prévu lorsque vous souhaitez stocker une date et une heure. MySQL affiche les valeurs de type DATETIME au format 'AAAA-MM-JJ HH:MM:SS'.

Pour les chaines de caractères :

Pour stocker un texte relativement court (moins de 255 caractères), vous pouvez utiliser les types CHAR et VARCHAR. Ces deux types s'utilisent avec un paramètre qui précise la taille que peut prendre votre texte (entre 1 et 255).

La différence entre CHAR et VARCHAR est la manière dont ils sont stockés en mémoire. Un CHAR(x) stockera toujours *x* caractères, en remplissant si nécessaire le texte avec des espaces vides pour le compléter, tandis qu'un VARCHAR(x) stockera jusqu'à *x* caractères (entre 0 et *x*), et stockera en plus en mémoire la taille du texte stocké.
Si vous entrez un texte plus long que la taille maximale définie pour le champ, celui-ci sera tronqué.

Pour stocker un text plus que 255 caractères, utilisez le type TEXT

Le type BLOB est également utilisé pour les gros fichiers

Le type ENUM

C'est un string avec un ensemble de valeurs.(il peut remplacer la contrainte CHECK dans certains cas)

Exemple :

```
create table prog(id enum ('inf','tge','tad'),
constraint pkpro primary key(id),
nom varchar(40));
```

Exemple :

```
CREATE TABLE PROGRAMMES
(
  CODEPRG  int unsigned,
  CONSTRAINT PKCODEPRG PRIMARY KEY(CODEPRG),
  NOMPROG VARCHAR(30)
);
```

Remarque

1. Pour donner un nom à la contrainte de clé primaire, il faut qu'elle soit au niveau table. (ce qui n'est pas nécessairement le même cas avec Oracle)

```
CREATE TABLE ETUDIANTS
(
 NUMAD int unsigned auto_increment,
 CONSTRAINT PKNUMAD PRIMARY KEY(numad),
 NOM VARCHAR(20) NOT NULL,
 PRENOM VARCHAR(20),
 CODEPRG int unsigned,
 VILLE VARCHAR(30),
 TELEPHONE VARCHAR(20),
  CONSTRAINT FKCODEPRG FOREIGN KEY(CODEPRG) REFERENCES
PROGRAMMES(CODEPRG)
);
```

Remarques :

1. Numad est un entier positif
2. Pas besoin de séquence pour une incrémentation auto,.
3. La clé étrangère se définie comme dans ORACLE.
4. Même si ce n'est pas dans l'exemple, la contrainte CHECK se définie comme dans ORACLE au niveau table seulement.

Et les procédures et fonctions?

À quelques exceptions, Le corps des procédures et fonctions sont identiques à ce que nous avons vu avec Oracle.

Certaines différences sont à considérer.

1. Il faut un delimiter spécial dans la procédure (fonction)

Exemple 1

```
delimiter $$
create function calculer() returns integer
begin
declare total integer;
select count(*) into total from etudiants;
return total;
end $$
```

L'appel de la fonction se fait ainsi (PAS DE DUAL)

```
select calculer() ;
```

Exemple 2

```
use labo1;
select compter1(420);
drop function compter1;
```

```
delimiter |
create function compter1(pcodep int) returns integer
begin
declare tot integer;
select count(*) into tot from etudiants
group by CODEPRG having CODEPRG = pcodep;
return tot;
end |
```

2. Il faut le mot réservé DECLARE
3. Le type de paramètre (IN ou OUT) est transmis en premier.
4. La fonction n'a pas besoin de type de paramètre puisque c'est toujours en IN
5. Remarquez le S du returnS
6. Pour les appels de fonctions, il n'y a pas de FROM dual

Exemple 3

```
delimiter |

CREATE  PROCEDURE insertion(in pcodtype int, in pdscription varchar(20))
BEGIN
insert into typelivre values(pcodtype,pdscription);
commit;
END|
```

L'appel de la procédure se fait :

```
call insertion (23,'Cinéma');
```

Exemple 4

```
delimiter |

CREATE  PROCEDURE afficher(in pcodep int)

BEGIN

select nom, prenom from etudiants where CODEPRG = pcodep;

END|
```

L'appel se fait :

```
call afficher(420);
```

Les triggers

Même principe qu'Oracle avec les restrictions suivantes :

1- Il n'y a pas de raise_application_error. Il faudra écrire une procédure pour l'insertion des messages erreurs dans une table, puis lire le contenu
2- On ne peut pas combiner plusieurs opérations DML dans un même trigger. Il faudra écrire un trigger pour chaque :
3- Il n' y a pas de OF dans le update pour préciser la colonne.
4- Il n'y a pas les deux points devant le NEWet le OLD
5- Le declare est à l'intérieur du BEGIN

Version Oracle :

```
create or replace trigger empsal
before insert or update on employesclg
for each row
declare sal number;
begin
select avg(salaireemp) into sal from employesclg
group by codedep having codedep =:new.codedep;
if :new.salaireemp is null
then :new.salaireemp:=sal;
end if;
end;
```

Version Mysql :

```
DELIMITER |;
create trigger empsal before insert on employes
for each row
begin
declare sal decimal;
select avg(salaire) into sal from employes
group by codep having codep =new.codep;
if new.salaire is null
then set new.salaire =sal;
end if;
end |;
```

Attention ! Il faudra écrire un autre trigger pour BEFORE UPDATE.

Il faudra un trigger par fichier SQL (comme pour Oracle)

Autre exemple (suppression en cascade)

```
DELIMITER |;
 CREATE TRIGGER suppresion
before  delete ON DEPARTEMENTS
for each row
BEGIN
  DELETE FROM EMPLOYES
  WHERE  EMPLOYES.CODEP = OLD.codep;
END |;
```

Avec Oracle :

```
CREATE  or REPLACE TRIGGER ctrlSalaire2
BEFORE  UPDATE OF salaire ON EMPLOYESBIDON
FOR EACH ROW
BEGIN
IF (:NEW.SALAIRE < :OLD.SALAIRE)
THEN RAISE RAISE_APPLICATION_ERROR(-20324,'LE SALAIRE NE PEUT PAS BAISSER');
END IF;
END;
```

Avec MYSQL.

1- Il faudra créer une procédure qui va contenir qui sera appelée par le trigger si jamais il y a une erreur. Le principe est d'arrêter le trigger en faisant une opération MYSQL interdite comme par exemple insérer des valeurs nulle dans une variable déclarée not null;
Dans l'exemple, la procédure lorsqu'elle est appelé fait les opérations suivantes :

- Crée une table RAISE_ERROR avec une colonne not null;
- Affecte les deux paramètres de la procédure en IN à deux variables session @error_code et @error_message. Évidement la valeur de ces paramètres est connue au moment de l'appel.
- Insère une valeur null dans la table RAISE_ERROR

@var_name désigne une variable session en MYQL

```
DELIMITER |
CREATE PROCEDURE raise_application_error(IN CODEER INTEGER, IN MESSAGE
VARCHAR(255))
BEGIN
  CREATE TEMPORARY TABLE IF NOT EXISTS RAISE_ERROR(F1 INT NOT NULL);
  SELECT CODEER, MESSAGE INTO @error_code, @error_message;
  INSERT INTO RAISE_ERROR VALUES(NULL);
END;|
```

2- Si la procédure est appelée cela veut dire que le trigger a provoqué une erreur. Pour récupérer le code erreur et la valeur du message, on crée une autre procédure qui va afficher le contenu des variables session de la procédure précédente.

```
DELIMITER |
CREATE PROCEDURE get_last_custom_error()
BEGIN
SELECT @error_code, @error_message;
END;|
```

3- Enfin écrire le trigger

```
DELIMITER |
CREATE TRIGGER ctrlSalaire
BEFORE  UPDATE  ON EMPLOYES
FOR EACH ROW
BEGIN
IF (NEW.SALAIRE < OLD.SALAIRE)
THEN CALL raise_application_error(100, 'le salaire ne doit pas baisser!');
END IF;
END;|
```

4- Tester le trigger :

1- update employes set salaire =12 where empno=1;

2- Un message erreur s'affiche: la colonne F1 cannot be null;

3- CALL get_last_custom_error(); va afficher votre message erreur

	@error_code	@error_message
▶	100	le salaire ne doit pas baisser!

5- C'est un peu long, mais MYSQL n'a pas la fonction RAISE_APPLICATION_ERROR.

Et l'optimisation de requête ?

En principe, lorsqu'une requête SQL est envoyée au SGBD, celui-ci établit un plan d'exécution. Le module se charge d'établir un plan d'exécution s'appelle Optimizer.

Le fonctionnement de l'Optimizer globalement similaire pour l'ensemble des SGBDs (Oracle et SQL Server), en utilisant les étapes suivantes :

1. Validation syntaxique
2. Validation sémantique
3. Utilisation éventuelle d'un plan précédemment produit
4. Réécriture/Simplification de la requête
5. Exploration des chemins d'accès et estimation des coûts.
6. Désignation du chemin le moins coûteux, génération du plan d'exécution et mise en cache de ce dernier.

Mais, le développeur de bases de données doit connaître certaines règles qui permettent d'optimiser l'exécution de requêtes. En voici quelques-unes de ces règles (qui ne sont pas nécessairement d'ans l'ordre).

- R1 : éviter le SELECT * : écrire plutôt le nom des colonnes dont vous avez besoin pour la requête.

- R2 : Créez des indexes sur les colonnes que vous utilisez dans la clause WHERE. Pour plus de performances, ces indexes doivent-être créés après l'insertion des données dans la table.

Rappel : Un index est un objet permettant d'accélérer l'accès aux données La création d'un index sur une clé primaire se fait automatiquement par le système. Pour créer un index sur une colonne autre que la clé primaire on utilise la commande CREATE INDEX.

```
create index indexnom on joueurs(nom);
```

```
create index indexnomprenom on joueurs(nom,prenom);
```

Si la colonne sur laquelle l'index est créé a la contraint UNIQUE alors vous pouvez créer un index UNIQUE.

```
Create UNIQUE  index indexAlias on joueurs(Pseudo);
```

Même si les indexes permettent d'accélérer les recherches, trop d'indexe a l'effet inverse (ralentit les recherches)

- ✓ Un index ne peut être créé que sur une table (pas une vue).
- ✓ Les index UNIQUE applique les contraintes d'unicité

- ✓ Ne jamais créer trop d'index
- ✓ Créer des index sur une colonne ayant une petite plage de valeurs inutiles
- ✓ Un index se créer sur maximum 16 colonnes.

- R3 : Lorsque c'est possible, utilisez le WHERE à la place du Having.
- R4 : Éviter les jointures dans le WHERE, utilisez plutôt le INNER JOIN.

Exemple :Select e.deptno, d.dname, e.ename
From (select deptno, ename from emp) e inner join (select deptno, dname from dept) d
On e.deptno = d.deptno;

Rq: Dans la clause FROM on respecte la règle R1.

- R5 : Lorsque c'est possible, utilisez une jointure à la place d'une sous-requête. Les jointures sont l'essentiel des SGBDRs alors ils sont optimisés pour l'écriture des jointures.
- R6 : Utilisez le SELECT count(*) à la place de compter les colonnes. (Select count(*) from etudiants à la place de SELECT count(numad) from etudiants
- R7 : Évitez les fourchettes < et >; utilisez le BETWEEN
- Si possible, utilisez le BETWEEN à la place du Like
- R8 : transformer l'INTERSECT en jointure
- R9 : Utilisez le IN à la place du ANY et le <> ALL en NOT IN
- R10 : Éviter la clause DISTINCT dans le SELECT sauf si c'est absolument nécessaire
- R11 : Ordonnez par nom des colonnes plutôt que par les numéros

Règles de sécurité sur MySQL (partie 1)

Comme tout SGBD, MySQL dispose d'un système de sécurité assez avancé, mais malheureusement non standardisé (exemple la notion de rôle n'existe pas dans MySQL).

Rôle du système de privilèges

La fonction première du système de privilèges de MySQL est d'authentifier les utilisateurs se connectant à partir d'un hôte donné, et de leur associer des privilèges sur une base de données comme SELECT, UPDATE, DELETE, CREATE VIEW, CREATE TABLE, etc….
Le système de droits de MySQL s'assure que les utilisateurs font exactement ce qu'ils sont supposés pouvoir faire dans la base de données. Lorsque vous vous connectez au serveur, votre identité est déterminée par l'hôte d'où vous vous connectez et le nom d'utilisateur que vous spécifiez. Le système donne les droits en fonction de votre identité et de ce que vous voulez faire

Le contrôle d'accès de MySQL se fait en deux étapes :

1. Le serveur vérifie que vous êtes autorisé à vous connecter.
2. En supposant que vous pouvez vous connecter, le serveur vérifie chaque requête que vous soumettez, pour vérifier si vous avez les droits suffisants pour l'exécuter. Par exemple, si vous sélectionnez des droits dans une table, ou effacez une table, le serveur s'assure que vous avez les droits de SELECT pour cette table, ou les droits de DROP respectivement

Si vos droits ont changé (par vous-mêmes ou bien par un administrateur), durant votre connexion, ces changements ne prendront peut-être effets qu'à la prochaine requête

Le serveur stocke les droits dans des tables de droits, situées dans la base MySQL. Il lit le contenu de ces tables en mémoire lorsqu'il démarre, et les relit dans différentes circonstances.

Quand les modifications de privilèges prennent-ils effets ?

Lorsque mysqld est lancé, toutes les tables de droits sont lues, et sont utilisées. Les modifications aux tables de droits que vous faites avec GRANT, REVOKE sont immédiatement prises en compte par le serveur.

Lorsque le serveur remarque que les tables de droits ont été modifiées, les connexions existantes avec les clients sont modifiées comme ceci :

- ✓ Les droits de table et colonnes prennent effet à la prochaine requête du client.
- ✓ Les droits de bases prennent effet à la prochaine commande USE nom_de_base
- ✓ Les droits globaux et les modifications de droits prennent effets lors de la prochaine connexion.

Les commandes GRANT et REVOKE (à partir de la version 3.22.11 de MySQL)

Ces commandes permettent à l'administrateur système de créer et supprimer des comptes utilisateur et de leur donner ou retirer des droits.
Les informations sur les comptes MySQL sont stockées dans la base MySQL

Les commandes GRANT et REVOKE sont utilisées pour contrôler les accès à MySQL. Ne donnez pas plus de droits que nécessaire. Ne donnez jamais de droits à tous les serveurs hôte

Les droits sont donnés à 4 niveaux :

Niveau global
Les droits globaux s'appliquent à toutes les bases de données d'un serveur. Ces droits sont stockés dans la table mysql.user

REVOKE ALL ON *.* retirera seulement les privilèges globaux.

Niveau base de données
Les droits de niveau de base de données s'appliquent à toutes les tables d'une base de données. Ces droits sont stockés dans les tables mysql.db et mysql.host

REVOKE ALL ON db.*
Retirera seulement les privilèges de base de données.

Niveau table
Les droits de table s'appliquent à toutes les colonnes d'une table. Ces droits sont stockés dans la table mysql.tables_priv

REVOKE ALL ON db.table
Retirera seulement les privilèges de table.

Niveau colonne
Les droits de niveau de colonnes s'appliquent à des colonnes dans une table. Ces droits sont stockés dans la table mysql.columns_priv

Tableau des Privillèges pour GRANT and REVOKE

Privilege	Context
CREATE	databases, tables, or indexes
DROP	databases, tables, or views
GRANT OPTION	databases, tables, or stored routines
LOCK TABLES	databases
REFERENCES	databases or tables
EVENT	databases
ALTER	tables
DELETE	tables
INDEX	tables
INSERT	tables or columns
SELECT	tables or columns
UPDATE	tables or columns
CREATE TEMPORARY TABLES	tables
TRIGGER	tables
CREATE VIEW	views
SHOW VIEW	views
ALTER ROUTINE	stored routines
CREATE ROUTINE	stored routines
EXECUTE	stored routines
FILE	file access on server host
CREATE TABLESPACE	server administration
CREATE USER	server administration
PROCESS	server administration

Privilege	Context
PROXY	server administration
RELOAD	server administration
REPLICATION CLIENT	server administration
SHOW DATABASES	server administration
SHUTDOWN	server administration
SUPER	server administration
ALL [PRIVILEGES]	server administration
USAGE	server administration (sans aucun droit)

Exemples:

En tant que ROOT

```
create user user1 identified by 'user1';
grant all on *.* to user1;
```

et surtout pas:

```
grant all privileges on *.* to user1 with grant option;
```

Après la création de l'utilisateur user1, il a tous les droits, donc de créer sa propre base de données. Éviter de donner ce droit à n'importe qui.
Conseils : si vous administrez un serveur MySQL, créer la base de données puis donner les droits sur celle-ci.

Toujours en tant que root : (le user2 aura le droit de créer et de gérer ses propres objets dans la base de données dbuser2.

```
create user user2 identified by 'user2';
create database dbuser2;
grant all on dbuser2.* to user2 ;
```

Si l'option with grant option est précisée alors il pourra donner les mêmes droits sur sa bd à un autre utilisateur.

Cette commande permet de donner tous les droits à ruby mais uniquement sur la table livres de la base de données labo1 du User root

```
grant all on labo1.livres to ruby;
```

Voici une façon plus intéressante d'attribuer les droits (on va les restreindre au maximum).

```
grant select on Gestion.departements to ruby;
grant insert on Gestion.departements to ruby;
grant update on Gestion.departements to ruby;
grant select on Gestion.abscences to ruby;
grant update(adresse) on Gestion.employes to ruby;
grant create view on Gestion.abscences to ruby;
```

Pour le update et le insert on peut préciser les colonnes autorisées.

```
grant insert(num,nom,prenom) on Gestion.employes to ruby;
```

On peut attribuer plusieurs droits en une seule commande.

```
grant create view , select, update(salaire)  on  Gestion.employes to ruby;
```

Pour enlever les droits, utilisez REVOKE.

```
revoke select on Gestion.departements from ruby;
revoke all on Gestion.departements from ruby;
```

PDO (PHP Data Object) par l'exemple

Il existe 3 façons d'accéder aux bases de données par PHP : Mysql qui utilise des requêtes mais pas de procédures stockées, Mysqli très performant puisque du côté serveur; elle est propre à MYSQL(un peu comme OCI de Oracle) et enfin PDO qui est du côté PHP (peut-être moins performant mais très standard puis que cette méthode est du côté application (un peu comme le JDBC)

PDO est donc une API d'accès aux données (n'importe qu'elle SGBD), orienté objet utilisé par PHP. Il est disponible avec les versions PHP 5 et plus.

PDO présente plusieurs avantages (mis à part qu'il est objet) il s'utilise avec les procédures stockées et les requêtes paramétrées ce qui permet de se protéger contre les injections SQL.

Les classes sont :

PDO : une instance de PDO représente la connexion à une base de données. Le plus souvent une seule instance de PDO par exécution de PHP. Cette classe contient entre autre les méthodes exec(), query() et prepare()

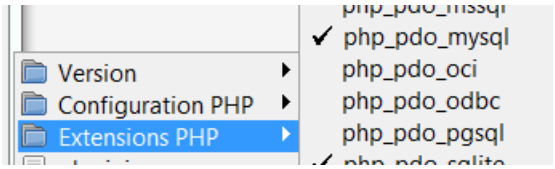
PDOStatement : une instance de PDOStatement représente une requête vers la base de données. Permet de préparer la requête puis de consulter son résultat. Cette classe contient entre autre les methodes fetch(), bindParmeter(), rowCount(),execute()

PDOException pour la gestion des Exceptions.

Étapes pour le traitement des commandes SQL :

Étape 0 : Activer PDO :

Normalement, PDO est activé par défaut. Pour le vérifier (voir la figure suivante), faites un clic gauche sur l'icône de WAMP dans la barre des tâches, puis allez dans le menu PHP / Extensions PHP et vérifiez que php_pdo_mysql est bien coché.



Étape 1 : Connexion à la base de données :

Comme tout SGBD, pour se connecter à la base de données, il faut avoir les paramètres suivants :

Nom du serveur ou son adresse IP (ou nom de l'hôte). C'est l'adresse de l'ordinateur où MySQL est installé. Le plus souvent, MySQL est installé sur le même ordinateur que PHP : dans ce cas, mettez la valeur localhost

Nom de la base de données sur laquelle vous voulez obtenir une connexion

Nom du user : nom de l'usager qui se connecte à la base de données.

Mot de passe du user : mot de passe de l'usager qui se connecte :

Syntaxe :

```
$connexion = new PDO('mysql:host=$hote;dbname=$basededonnee;charset=utf8', $user, $mpasse);
```

Exemple: (ici le user est le root, et il n' a pas de mot de passe)

```
$mybd = new PDO('mysql:host=localhost;dbname=animalrootmyadm;charset=utf8', 'root', '');
```

Ou encore:

```
try
{
$mybd = new PDO('mysql:host=localhost;dbname=animalrootmyadm;charset=utf8', 'root', '');
}
catch (PDOException $e)
{
        echo('Erreur de connexion: ' . $e->getMessage());
        exit();
}
```

Certains sites vous proposerons la fonction die() qui est équivalente à exit();

```
catch (Exception $e)
{die('Erreur : ' . $e->getMessage());}
```

Pour fermer la connexion :

```
$mybd=null;
```

Étape 2 : envoyer des requêtes à la base de données :

Cas1 : Requêtes exécutées une seule fois

La méthode **PDO ::exec() :** Cette méthode permet d'envoyer une requête de type DML à la base de données. Elle retourne un entier indiquant le nombre de lignes affectées

```
public int PDO::exec ( string $statement )
```

```
<?php
try
{
        $mybd = new PDO('mysql:host=localhost;dbname=animalrootmyadm;charset=utf8',
'root', '');
   echo('connexion reussie');
    echo ('insertion');
   $insertion = $mybd->exec("INSERT INTO chats(nom, race) VALUES ('ram','domestique')");
    echo('total insertion est ' . $insertion);
   echo('suppression');
   $suppresion = $mybd->exec("delete from chats where race ='ros'");
    echo('total suppression est ' . $suppresion);
 }
  catch (PDOException $e)
{
     echo('Erreur de connexion: ' . $e->getMessage());
     exit();
}
$mybd=null;
?>
```

La méthode PDO :**query()** Cette méthode permet d'envoyer une requête SELECT sans paramètres à la base de données. Elle retourne un PDOStatement (ou un curseur)

public PDOStatement PDO::query (string $statement)

Pour lire le curseur on utilise la méthode **fetch()** (méthode sql ou PL/SQL) et évidement une boucle while. Exemple :

while ($donnees = $curseur->fetch())

{ affichage }

La commande FETCH permet de lire ligne par ligne le contenu du curseur. À chaque fois que cette commande est appelée, le curseur avance au prochain enregistrement dans l'ensemble actif. Le résultat de la lecture est envoyé le table de nom donnees

Exemple 1 (fetch par défaut)

```
<?php
try
{
$mybd = new PDO('mysql:host=localhost;dbname=animalrootmyadm;charset=utf8', 'root', '');
    echo('connexion reussie') . '<br />';
   echo ('affichage du contenu de la table chats') . '<br />';
      $resultat = $mybd->query("SELECT * FROM CHATS limit 5");
    while ($donnees = $resultat->fetch())
{
echo $donnees[1] . $donnees[2] .'<br />';
}
echo ($resultat->rowCount());// nombre totale de lignes
$resultat->closeCursor();
}
   catch (PDOException $e)
{      echo('Erreur de connexion: ' . $e->getMessage()); exit(); }
```

```
$mybd=null;
?>
```

Dans l'exemple précédent, la lecture du tableau de données se fait par l'indice de colonne. L'indice de la première colonne est **ZÉRO.**

Parfois, il est nécessaire de dire à votre programme comment vous voulez lire vos données. Comme par exemple accéder au tableau par le nom des colonnes.

PDO::FETCH_ASSOC: retourne un tableau indexé par le nom des colonne;

PDO::FETCH_OBJ: retourne un objet dont les propriétés correspondent aux nom de colonnes.

Exemple 2(FETCH_ASSOC**)**

```
<?php
try
{
$mybd = new PDO('mysql:host=localhost;dbname=animalrootmyadm;charset=utf8', 'root', '');
   echo('connexion reussie') . '<br />';
  echo ('affichage du contenu de la table chats') . '<br />';
    $resultat = $mybd->query("SELECT * FROM CHATS limit 8");
  $resultat->setFetchMode(PDO::FETCH_ASSOC);
   while ($donnees = $resultat->fetch())
{
  echo $donnees['nom'] . $donnees['race'] .'<br />';
}
echo ($resultat->rowCount());// nombre totale de lignes
$resultat->closeCursor();
}
```

```
 catch (PDOException $e)
{echo('Erreur de connexion: ' . $e->getMessage()); exit();}
$mybd=null;
?>
```

Exemple 3(FETCH_OBJ)

```
<?php
try
{
$mybd = new PDO('mysql:host=localhost;dbname=animalrootmyadm;charset=utf8', 'root', '');
    echo('connexion reussie') . '<br />';
   echo ('affichage du contenu de la table chats') . '<br />';
    $resultat = $mybd->query("SELECT * FROM CHATS limit 3");
   $resultat->setFetchMode(PDO::FETCH_OBJ);
    while ($donnees = $resultat->fetch())
{
echo $donnees->nom . $donnees->race .'<br />';
}
echo ($resultat->rowCount());// nombre totale de lignes
$resultat->closeCursor();
}
```

Cas 2 : Requêtes avec paramètres :

Les requêtes paramétrées sont incontournables lorsque nous essayons d'accéder à une base de données via le web. Elles ont un avantage majeur qui est de réduire considérablement les injections SQL. De plus ce sont des requêtes préparées donc exécutées plus d'une fois et précompilées.

Comme en JDBC, en PHP les paramètres peuvent être représentés par le **?** (Le point d'interrogation).

Et qu'allons-nous utiliser pour les requêtes paramétrées? Le PDOStatement (qui ressemble au PreparedStatemnt de JDBC).

Le fonctionnement de l'exécution de requêtes paramétrées est :

La methode **prepare() :** cette méthode de la classe PDO permet de passer une requête paramétrée au SGBD.

public PDOStatement **PDO::prepare** (string $statement)

La methode BindParam() de la classe PDOStatement permet de lier les variables aux valeurs.

public bool **PDOStatement::bindParam** ($parameter , &$variable [, int $data_type = PDO::PARAM_STR [, int $length]])

Exemples:

$stmt1->bindParam(1, $nom);

$stm->bindParam(2, $race, PDO::PARAM_STR);

$stm->bindParam(2, $race, PDO::PARAM_STR,20);

La méthode execute() permet d'exécuter la requête avec ses paramètres. Elle retourne True ou false selon le succès de l'exécution de la requête

public bool **PDOStatement::execute** ([array `$input_parameters`])

Exemple 1

```
 <?php
try
{
$mybd = new PDO('mysql:host=localhost;dbname=animalrootmyadm;charset=utf8', 'root', '');
    echo('connexion reussie') . '<br />';
   echo ('insertion avec paramètres') . '<br />';
$stmt1 = $mybd->prepare("INSERT INTO chats(nom, race) VALUES (?, ?)");
//on fait un bind sur les paramètres par l'indice d'apparition dans la requête.
$stmt1->bindParam(1, $nom);
$stmt1->bindParam(2, $race);
// affectation de valeurs aux paramètres
$nom = 'Kally';
$race = 'Bingale';
// executer la requête
$total= $stmt1->execute();
echo('total insertion est ' . $total);
}
   catch (PDOException $e)
{
      echo('Erreur de connexion: ' . $e->getMessage());
```

```
    exit();
}
$mybd=null;
?>
```

Vous pouvez remplacer les ? par **:nomPrametre** (deux points suivi du nom de paramètre

Exemple 2:

```
<?php
try
{
$mybd = new PDO('mysql:host=localhost;dbname=animalrootmyadm;charset=utf8', 'root', '');
   echo('connexion reussie') . '<br />';
    echo ('insertion avec paramètres') . '<br />';
  $stmt1 = $mybd->prepare("INSERT INTO chats(nom, race) VALUES (:pnom,:prace)");
// on fait un bind sur les noms des paramètres
$stmt1->bindParam(':pnom', $nom);
$stmt1->bindParam(':prace', $race);
// Affectation de valeurs aux paramètres
$nom = 'BeauBeau';
$race = 'domestique';
// executer la requête
$total= $stmt1->execute();
echo('total insertion est ' . $total);
}
  catch (PDOException $e)
{echo('Erreur de connexion: ' . $e->getMessage()); exit();}
  $mybd=null;
```

```
?>
```

Exemple 3 (avec un select)

```
 <?php
try
{
$mybd = new PDO('mysql:host=localhost;dbname=exercice1;charset=utf8', 'ruby',
'ruby2016');
   echo('connexion reussie') . '<br />';
  $stmt1 = $mybd->prepare("select * from chats where race  like ? ;)");
// affectation de valeurs aux paramètres
$race = 'Abyssin';
// Liaison des paramètre
$stmt1->bindParam(1, $race, PDO::PARAM_STR, 20);
// executer la requête
 $stmt1->execute();
  while ($donnees = $stmt1->fetch())
{
  echo $donnees[1] . $donnees[2] .'<br />';
}
echo ($stmt1->rowCount());
$stmt1->closeCursor();
```

```
}
  catch (PDOException $e)
{ echo('Erreur de connexion: ' . $e->getMessage());
     exit();}
 $mybd=null;
?>
```

Cas3 : Appel de procédures.

Lorsqu'on appelle une procédure stockée on procède de la même manière qu'une requête paramétrée. On utilise le PDO Statement avec le la méthode prepare() sauf qu'au lieu qu'elle prenne comme paramètre la requête, on utilise un CALL de la procédure.

Il faudra, en plus de faire le bind des paramètres en IN, préciser le type des paramètres en OUT.

Exemple 1 : procédure qui retourne un ensemble de résultats (donc un curseur).

Voici la procédure qui sera appelée dans l'exemple qui va suivre

```
delimiter |
CREATE  PROCEDURE ListeChats(in race1 varchar(10))
begin
select * from chats
where race = race1;
end
```

Remarque :

PDOStatement::bindParam: Associe une valeur à un paramètre

public bool **PDOStatement::bindParam** ($parameter , $value ,data_ype)

Parameter :Identifiant du paramètre. Pour une requête préparée ou une peocédure stockée. Le paramètre est l'indice du ? ou :NomParame

indexé numériquement qui commence à la position 1 du paramètre.

Value : La valeur à associer au paramètre.

data_type : le type du paramètre :

Exemple :

$stm->bindParam(1, $race, PDO::PARAM_STR,10);

Indique que le paramère 1 reçoit le contenue de la variable race. Ce paramètre est de type string de longueur 10

```
<?php
try
{
$mybd = new PDO('mysql:host=localhost;dbname=animalrootmyadm;charset=utf8', 'root', '');
$stm = $mybd->prepare("CALL ListeChats(?)", array(PDO::ATTR_CURSOR,
PDO::CURSOR_FWDONLY));
$stm->bindParam(1, $race);
   $race="Bingale";
 // $stm->bindParam(1, $race, PDO::PARAM_STR,10);
  $stm->execute();
 //while ($donnees = $stm->fetch())
//while ($donnees = $stm->fetch(PDO::FETCH_ASSOC, PDO::FETCH_ORI_NEXT))
while ($donnees = $stm->fetch(PDO::FETCH_NUM, PDO::FETCH_ORI_NEXT))
{
          echo $donnees[1] . '<br />';
}
$stm->closeCursor();
}
catch (PDOException $e)
{
```

```
    echo('Erreur de connexion: ' . $e->getMessage());
    exit();
     }
$mybd=null;
?>
```

Exemple 2, appel d'une procédure insertion :

```
delimiter |
CREATE  PROCEDURE insertChat(in pnom varchar(20), in prace varchar(20))
BEGIN
insert into chats (nom,race)values(pnom,prace);
commit;
END |
```

L'appel de la procédure:

```
<?php
try
{
$mybd = new PDO('mysql:host=localhost;dbname=animalrootmyadm;charset=utf8', 'root', '');
  $stm = $mybd->prepare("CALL insertChat(?,?)");
    $stm->bindParam(1, $nom,PDO::PARAM_STR,20);
  $stm->bindParam(2, $race);
   $nom ="Gribouille";
   $race="Lynx";
  $total = $stm->execute();
 echo('total insertion est ' . $total);
}
catch (PDOException $e)
{ echo('Erreur de connexion: ' . $e->getMessage()); exit(); }
```

```
$mybd=null;
?>
```

Exemple 3, appelle d'une fonction qui retourne un entier : L'appel d'une fonction se fait comme l'appel de procédure. À la place d'utiliser CALL, vous utiliserez le SELECT

```
delimiter |
create function CompterChat(prace varchar(20)) returns integer
DETERMINISTIC
begin
declare total integer;
select count(*) into total from chats where race=prace;
return total;
end |
```

```
<?php
try
{
$mybd = new PDO('mysql:host=localhost;dbname=animalrootmyadm;charset=utf8', 'root', '');
  $stm = $mybd->prepare("SELECT CompterChat(?)" );
$stm->bindParam(1, $race);
  $race="Bingale";
   $stm->execute();
 if ($donnees = $stm->fetch())
{
          echo $donnees[0] . '<br />';
}
$stm->closeCursor();
```

```
}
catch (PDOException $e)
{ echo('Erreur de connexion: ' . $e->getMessage());exit();}
$mybd=null;
?>
```

Compléments

PDOStatement::fetch — Récupère la ligne suivante d'un jeu de résultats PDO

Description ¶
public mixed **PDOStatement::fetch** ([int $fetch_style [, int $cursor_orientation = PDO::FETCH_ORI_NEXT [, int $cursor_offset = 0]]])

Récupère une ligne depuis un jeu de résultats associé à l'objet *PDOStatement*. Le paramètre fetch_style détermine la façon dont PDO retourne la ligne.

Liste de paramètres ¶
fetch_style : Contrôle comment la prochaine ligne sera retournée à l'appelant. Cette valeur doit être une des constantes *PDO::FETCH_ **, et par défaut, vaut la valeur de *PDO::ATTR_DEFAULT_FETCH_MODE* (qui vaut par défaut la valeur de la constante *PDO::FETCH_BOTH*).

- *PDO::FETCH_ASSOC*: retourne un tableau indexé par le nom de la colonne comme retourné dans le jeu de résultats
- *PDO::FETCH_BOTH* (défaut): retourne un tableau indexé par les noms de colonnes et aussi par les numéros de colonnes, commençant à l'index 0, comme retournés dans le jeu de résultats
- *PDO::FETCH_BOUND*: retourne **TRUE** et assigne les valeurs des colonnes de votre jeu de résultats dans les variables PHP à laquelle elles sont liées avec la méthode PDOStatement::bindColumn()
- *PDO::FETCH_CLASS*: retourne une nouvelle instance de la classe demandée, liant les colonnes du jeu de résultats aux noms des propriétés de la classe. Si fetch_style inclut PDO::FETCH_CLASS (c'est-à-dire *PDO::FETCH_CLASS | PDO::FETCH_CLASSTYPE*), alors le nom de la classe est déterminé à partir d'une valeur de la première colonne.
- *PDO::FETCH_INTO* : met à jour une instance existante de la classe demandée, liant les colonnes du jeu de résultats aux noms des propriétés de la classe

- *PDO::FETCH_LAZY* : combine *PDO::FETCH_BOTH* et *PDO::FETCH_OBJ*, créant ainsi les noms des variables de l'objet, comme elles sont accédées
- *PDO::FETCH_NAMED* : retourne un tableau de la même forme que *PDO::FETCH_ASSOC*, excepté que s'il y a plusieurs colonnes avec les mêmes noms, la valeur pointée par cette clé sera un tableau de toutes les valeurs de la ligne qui a ce nom comme colonne
- *PDO::FETCH_NUM* : retourne un tableau indexé par le numéro de la colonne comme elle est retourné dans votre jeu de résultat, commençant à 0
- *PDO::FETCH_OBJ* : retourne un objet anonyme avec les noms de propriétés qui correspondent aux noms des colonnes retournés dans le jeu de résultats

cursor_orientation :Pour un objet PDOStatement représentant un curseur scrollable, cette valeur détermine quelle ligne sera retournée à l'appelant. Cette valeur doit être une des constantes *PDO::FETCH_ORI_**, et par défaut, vaut *PDO::FETCH_ORI_NEXT*. Pour demander un curseur scrollable pour votre objet PDOStatement, vous devez définir l'attribut *PDO::ATTR_CURSOR* à *PDO::CURSOR_SCROLL* lorsque vous préparez la requête SQL avec la fonction PDO::prepare().

Offset : Pour un objet PDOStatement représentant un curseur scrollable pour lequel le paramètre *cursor_orientation* est défini à *PDO::FETCH_ORI_ABS*, cette valeur spécifie le numéro absolu de la ligne dans le jeu de résultats qui doit être récupérée.

Pour un objet PDOStatement représentant un curseur scrollable pour lequel le paramètre *cursor_orientation* est défini à *PDO::FETCH_ORI_REL*, cette valeur spécifie la ligne à récupérer relativement à la position du curseur avant l'appel à la fonction **PDOStatement::fetch()**.

Valeurs de retour : La valeur retournée par cette fonction en cas de succès dépend du type récupéré. Dans tous les cas, **FALSE** est retourné si une erreur survient.

La classe PDOStatement

- PDOStatement->bindColumn — Lie une colonne à une variable PHP
- PDOStatement->bindParam — Lie un paramètre à un nom de variable spécifique
- PDOStatement->bindValue — Associe une valeur à un paramètre
- PDOStatement->closeCursor — Ferme le curseur, permettant à la requête d'être de nouveau exécutée
- PDOStatement->columnCount — Retourne le nombre de colonnes dans le jeu de résultats
- PDOStatement->debugDumpParams — Détaille une commande préparée SQL
- PDOStatement->errorCode — Récupère le SQLSTATE associé lors de la dernière opération sur la requête

- PDOStatement->errorInfo — Récupère les informations sur l'erreur associée lors de la dernière opération sur la requête
- PDOStatement->execute — Exécute une requête préparée
- PDOStatement->fetch — Récupère la ligne suivante d'un jeu de résultat PDO
- PDOStatement->fetchAll — Retourne un tableau contenant toutes les lignes du jeu d'enregistrements
- PDOStatement->fetchColumn — Retourne une colonne depuis la ligne suivante d'un jeu de résultats
- PDOStatement->fetchObject — Récupère la prochaine ligne et la retourne en tant qu'objet
- PDOStatement->getAttribute — Récupère un attribut de requête
- PDOStatement->getColumnMeta — Retourne les métadonnées pour une colonne d'un jeu de résultats
- PDOStatement->nextRowset — Avance à la prochaine ligne de résultats d'un gestionnaire de lignes de résultats multiples
- PDOStatement->rowCount — Retourne le nombre de lignes affectées par le dernier appel à la fonction PDOStatement::execute()
- PDOStatement->setAttribute — Définie un attribut de requête
- PDOStatement->setFetchMode — Définit le mode de récupération par défaut pour cette requête

La classe PDO

- PDO::beginTransaction — Démarre une transaction
- PDO::commit — Valide une transaction
- PDO::__construct — Crée une instance PDO qui représente une connexion à la base
- PDO::errorCode — Retourne le SQLSTATE associé avec la dernière opération sur la base de données
- PDO::errorInfo — Retourne les informations associées à l'erreur lors de la dernière opération sur la base de données
- PDO::exec — Exécute une requête SQL et retourne le nombre de lignes affectées
- PDO::getAttribute — Récupère un attribut d'une connexion à une base de données
- PDO::getAvailableDrivers — Retourne la liste des pilotes PDO disponibles
- PDO::inTransaction — Vérifie si nous sommes dans une transaction
- PDO::lastInsertId — Retourne l'identifiant de la dernière ligne insérée ou la valeur d'une séquence
- PDO::prepare — Prépare une requête à l'exécution et retourne un objet
- PDO::query — Exécute une requête SQL, retourne un jeu de résultats en tant qu'objet PDOStatement
- PDO::quote — Protège une chaîne pour l'utiliser dans une requête SQL PDO
- PDO::rollBack — Annule une transaction
- PDO::setAttribute — Configure un attribut PDO

Injections SQL

Rappels : (mêmes principes en JDBC avec oracle→session 4)

L'injection SQL directe est une technique où un pirate modifie une requête SQL existante pour afficher des données cachées, ou pour écraser des valeurs importantes, ou encore exécuter des commandes dangereuses pour la base. Cela se fait lorsque l'application prend les données envoyées par l'internaute, et l'utilise directement pour construire une requête SQL. C'est Exploiter les failles de sécurité par de personnes non autorisées et malintentionnées. On en parle plus pour les applications Web

Comment s'en prévenir en général ?

1. Valider toutes les entrées. Vérifier que les données ont bien les types attendus
2. Vérifier le format des données saisies et notamment la présence de caractères spéciaux
3. Ne pas afficher de messages d'erreur explicites affichant la requête ou une partie de la requête SQL. Personnalisez vos messages erreur.
4. Supprimer les comptes utilisateurs non utilisés, notamment les comptes par défaut (Anonymous, Test,)
5. Éviter les comptes sans mot de passe
6. Restreindre au minimum les privilèges des comptes utilisés.(un utilisateur adapté avec des droits très limités)
7. Utiliser des comptes différents pour accéder votre serveur distant
8. Ne vous connectez jamais à la BD avec un compte de super utilisateur (root)
9. Les procédures stockées (systèmes ou non) non nécessaires à l'application doivent être supprimées ou avec un accès restreint.
10. ET surtout utiliser des variables de liaison (requêtes paramétrées ou procédures stockées)

Sources

http://php.net/manual/fr/book.pdo.php

http://nl3.php.net/manual/fr/pdostatement.fetch.php

http://downloads.mysql.com/docs/refman-5.0-fr.pdf

http://php.net/manual/fr/pdostatement.bindvalue.php

https://dev.mysql.com/doc/refman/5.7/en/stored-programs-logging.html

http://fmaz.developpez.com/tutoriels/php/comprendre-pdo/#LIII.c

http://php.net/manual/fr/security.database.sql-injection.php

https://blogs.oracle.com/svetasmirnova/entry/how_to_raise_error_in