Comprendre PDO

par Francois Mazerolle (http://www.maz-concept.com)

Date de publication : 28 Octobre 2010

Dernière mise à jour :

De ma perspective personnelle, il m'a semblé que beaucoup de développeurs PHP soient rebutés par PDO. Je dois admettre qu'au tout début, j'étais moi-même dubitatif, et la plupart des tutoriels disponibles m'expliquait comment faire, mais jamais clairement pourquoi le faire. Ensemble, nous verrons donc quel est le rôle de PDO, en comparaison à mysql_ et mysqli_.

I - Introduction....3
I.a - Tous ne sont qu'extensions....3
I.b - Qu'est-ce que PDO ?....3
I.c - Pourquoi PDO plutôt qu'un autre ?....4
I.d - Encore en développement....4
II - Les requêtes préparées....5
II.a - Qu'est-ce qu'une requête préparée ?....5
II.b - PDO != requête préparée....6
II.c - Pourquoi les requêtes préparées ?....6
II.d - Les requêtes préparées sont plus sécurisées, vraiment ?....6
III - La pratique, utiliser PDO....6
III.a - Établir une connexion avec PDO....6
III.b - Sans les requêtes préparées....7
III.c - Avec les requêtes préparées....8
III.d - Réutiliser une requête préparée pour gagner en performance....8
IV - Les place holders....9
IV.a - Nommage des place holders....9
IV.b - Comment faire face à un nombre de place holders dynamique ?....9
V - Autres cas particuliers....10
V.a - Trouver le nombre de lignes retournées....10
V.b - Trouver le nombre de lignes affectées....11
V.c - Explication du faux bogue de la clause LIMIT....11
VI - Conclusion....12

I - Introduction

I.a - Tous ne sont qu'extensions

Saviez-vous que les fonctions ayant le préfixe mysql_ font partie d'une extension nommée mysql ? Eh oui, il en va de même avec mysqli_... et PDO aussi d'ailleurs ! À l'heure actuelle, ce sont les trois API disponibles pour établir une connexion à un serveur MySQL avec PHP.

Maintenant que vous avez bien compris qu'il existe trois extensions permettant à PHP de se connecter à MySQL, voici un tableau comparatif :

	Extension MySQL	Extension mysqli	PDO (avec le pilote PDO MySQL Driver et MySQL Native Driver)
Version d'introduction en PHP	Avant 3.0	5.0	5.0
Inclus en PHP 5.x	Oui, mais désactivé	Oui	Oui
Statut de développement MySQL	Maintenance uniquement	Développement actif	Développement actif depuis PHP 5.3
Recommandée pour les nouveaux projets MySQL	Non	Oui	Oui
L'API supporte les jeux de caractères	Non	Oui	Oui
L'API supporte les commandes préparées	Non	Oui	Oui
L'API supporte les commandes préparées côté client	Non	Non	Oui
L'API supporte les procédures stockées	Non	Oui	Oui
L'API supporte les commandes multiples	Non	Oui	La plupart
L'API supporte toutes les fonctionnalités MySQL 4.1 et plus récent	Non	Oui	La plupart

Source : ***http://www.php.net/manual/fr/mysqli.overview.php***

Ce tableau démontre principalement une chose : l'extension MySQL à fait son temps.

I.b - Qu'est-ce que PDO ?

PDO signifie **P**hp **D**ata **O**bject. Il s'agit une couche d'abstraction des fonctions d'accès aux bases de données. Ça ne signifie donc pas que vos requêtes seront automatiquement compatibles avec n'importe quelle base de données, mais bien que les fonctions d'accès seront universelles :
mysql_connect(), mysql_query(), mysql_result(), mysql_fetch_array(), mysql_real_escape_string(), etc.
... toutes ces fonctions que vous utilisez sont spécifiques à un SGBD et leur utilisation sera désormais automatiquement déterminée par PDO.

Ainsi, il est temps de casser le mythe laissant sous-entendre que PDO permet une compatibilité entre plusieurs types de bases de données ; PDO n'a pas pour but que d'interpréter vos requêtes et de les traduire pour tous les SGBD. Le but premier de PDO est surtout d'éviter d'apporter une solution au problème de code tel que celui-ci :

Sans PDO
```
switch($typeDb)
{
    case 'mysql':
        mysql_query($query);
        break;
    case 'sqlite':
         sqlite_query($query);
         break;
    case 'mssql':
         mssql_query($query);
         break;
    case 'oci':
         $stid = oci_parse($conn, $query);
         oci_execute($stid);
    //etc...
}
```

En somme, lorsque vous démarrez une connexion PDO, il faudra indiquer quel est le type de SGDB à utiliser. Ensuite, il suffira d'utiliser une fonction unique, fournie par PDO. Pour faire une image, PDO s'occupera de diriger votre requête « vers une fonction d'accès appropriée ». Ça, c'est PDO.

I.c - Pourquoi PDO plutôt qu'un autre ?

Il existe des solutions alternatives à PDO. MDB2, de Pear, en est un bon exemple. La différence majeure est que PDO est une extension compilée de PHP et offre donc des performances supérieures dans la plupart des cas. De plus, des rumeurs fortement insistantes et assez crédibles laissent croire que les futures évolutions de PHP délaisseront progressivement les fonctions standards pour proposer PDO comme solution d'accès aux SGBD par défaut:

Déplacement des extensions de bases de données non PDO vers PECL
[...] Afin de pousser à l'utilisation de cette interface commune, les vieilles extensions PHP seront déplacées vers PECL. Ainsi il ne sera plus possible de faire appel à mysql_connect() avec le paquet d'installation par défaut.
Source : http://www.phpteam.net/index.php/articles/les-nouveautes-de-php-6
D'ailleurs depuis la version 5 de PHP :

PHP 5+ : MySQL n'est plus activé par défaut [...].
Source : http://www.php.net/manual/fr/mysql.installation.php

I.d - Encore en développement

Vous vous souvenez des cases "La plupart" que PDO avait dans le tableau de l'introduction ? Eh bien oui, c'est que PDO est encore en développement. Bien que PDO ne soit pas nouveau, son utilisation est encore peu répandue. Il est vrai que certains connecteurs (ou pilotes) de PDO sont encore en développement, et que d'autres n'existent pas encore. Voici une **liste des connecteurs disponibles**, en date du 25 juin 2010 :

- IBM
- Informix
- MySQL
- ODBC et DB2
- Postgre SQL
- SQLite

Les connecteurs suivants sont aussi disponibles, mais expérimentaux :

- 4D
- FireBird/Interbase
- MS SQL (je suis personnellement incapable de transférer des données unicode)
- Oracle

Chaque connecteur peut être ou non activé sur votre serveur sous forme d'extension. Sous Windows, on retrouve donc une DLL pour PDO, en plus d'une DLL pour chaque connecteur. Sous Linux, ce sont des modules .so, mais le principe est le même.

Si vous n'êtes pas certain, les fonctions **PDO::getAvailableDrivers**() et **phpinfo**(), pourront vous indiquer si PDO est installé sur votre serveur, et quels sont les connecteurs disponibles.

II - Les requêtes préparées

II.a - Qu'est-ce qu'une requête préparée ?

Les requêtes préparées ne sont pas propres à PDO, il est possible d'en faire autant avec mysql_ (*voir note) qu'avec mysqli_. Le concept est de soumettre un moule de requête et de placer des *place holders* aux endroits où l'on voudra insérer nos valeurs dynamiques. Attention, j'ai bien dit valeurs, et non pas noms de tables, noms de champs, ni même commandes ou constantes internes (DESC, ASC, etc.). Un place holder représente donc une seule et unique valeur.

** En soi, les fonctions mysql_ n'offrent aucune fonction supportant les requêtes préparées. Malgré tout, nous verrons qu'il est possible d'en effectuer manuellement depuis toujours (enfin, depuis la version 4.1 de MySQL...).*

Voici le principe, vous avez une requête avec des *place holders* (les points d'interrogation) :

```
SELECT *
FROM foo
WHERE id=? AND bar<?
LIMIT ?;
```

Le SGBD va préparer (interpréter, compiler et stocker temporairement son "plan d'exécution logique" (merci doctorrock)) la requête.

Il faudra ensuite associer des valeurs aux *place holders*, qui agissent un peu comme des variables :

```
place holder #1 = 1
place holder #2 = 100
place holder #3 = 3
```

Nous verrons le véritable code pour associer les valeurs plus bas.
Puis, dans une seconde phase, ces valeurs seront soumises en demandant la compilation et l'exécution de la requête qui à été préparée. Le SGDB saura alors que ce qu'elle insère, ce sont des valeurs et non pas des commandes. L'assemblage de la requête, ou sa compilation finale, se fera en interne du SGBD, ce qui explique que vous ne puissiez pas réellement déboguer la valeur compilée de votre requête dans votre code PHP. Dans ce cas-là, la requête exécutée sera donc :

```
SELECT *
FROM foo
WHERE id=1 AND bar<100
LIMIT 3;
```

II.b - PDO != requête préparée

Une des possibilités de PDO, qui est extrêmement populaire, est l'utilisation des requêtes préparées. Beaucoup de personnes semblent confondre cette approche et PDO lui-même. Comme je l'ai affirmé précédemment, PDO a pour but d'offrir une abstraction des fonctions d'accès. En soi, il n'impose pas l'utilisation des requêtes préparées. Comme nous le verrons, PDO n'est pas non plus le seul à permettre les requêtes préparées.

Quoi qu'il en soit, PDO rend l'utilisation des requêtes préparées extrêmement intéressante et accessible, tellement que la plupart des tutoriels passent directement de l'exécution des requêtes avec mysql_query(), aux multiples lignes requises pour exécuter une requête préparéee dans PDO. Cet article sera certes plus long, mais nous prendrons ensemble le temps de bien faire le pont entre les deux approches...

II.c - Pourquoi les requêtes préparées ?

Soyons bien honnête :

- il vous faudra écrire plus de lignes que pour une requête simple ;
- pour une exécution unique, les performances sont moindres qu'avec une exécution directe ;
- le débogage d'une requête est légèrement plus complexe ;
- et concrètement, le résultat est, à toutes fins pratiques, identique : exécuter une requête.

Alors pourquoi diable a-t-on inventé les requêtes préparées ? Eh bien, exactement pour la même raison que nous avons inventé les systèmes de templates : pour isoler les données du traitement.

Ainsi, l'utilisation de requêtes préparées offrira les avantages suivants :

- impose une certaine rigueur de programmation ;
- optimisation du temps d'exécutions requis pour les requêtes exécutées plus d'une fois ;
- plus grande sécurité au niveau des requêtes.

II.d - Les requêtes préparées sont plus sécurisées, vraiment ?

Sans prétendre que de ne pas utiliser de requêtes préparées soit un risque, l'argument de sécurité demeure très important. Au sommet du top 10 des risques de sécurité dans les applications web de l'OWASP pour l'année 2010, se trouvent les failles d'injection. Et la solution recommandée est :

Preventing injection requires keeping untrusted data separate from commands and queries.

Source : http://www.owasp.org/index.php/Top_10_2010-Injection

Ce qui peut se traduire par : « Prévenir les injections requiert de séparer les données non sûres des commandes et requêtes. »

Et ça tombe bien ; c'est très précisément ce que font les requêtes préparées : séparer les données de la structure de la requête !

III - La pratique, utiliser PDO

III.a - Établir une connexion avec PDO

La première chose à faire pour utiliser PDO est bien sûr d'établir une connexion à une base de données :

Établir une connexion
```
$strConnection = 'mysql:host=localhost;dbname=ma_base';
```

Établir une connexion
```
$arrExtraParam= array(PDO::MYSQL_ATTR_INIT_COMMAND => "SET NAMES utf8");
$pdo = new PDO($connStr, 'Utilisateur', 'Mot de passe', $arrExtraParam); //Instancie la connexion
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
```

La première ligne définit la chaine de connexion (DSN). Il s'agit d'une syntaxe plutôt particulière, débutant par le type de connecteur à utiliser. Ici, mysql: indique à PDO d'utiliser son connecteur MySQL. Cette ligne contient aussi les deux autres paramètres : l'adresse du serveur de bases de données, ainsi que la base de données à utiliser.

À la seconde ligne, nous demandons l'exécution d'une commande pour l'utilisation de l'UTF-8. Nous aurions aussi pu faire une requête standard après l'instanciation de PDO. Vous noterez au passage qu'il est possible de passer les paramètres de connexion via le quatrième paramètre du constructeur, ou par la suite, via la fonction setAttribute().

La troisième ligne est le véritable début de notre aventure : l'instanciation de la connexion. Si la connexion échoue, PDO lancera une exception.

Finalement, la dernière ligne demande de rapporter les erreurs sous forme d'exceptions. Par défaut, PDO est configuré en mode silencieux; il ne rapportera pas les erreurs. Il existe trois modes d'erreurs :

- **PDO::ERRMODE_SILENT** - ne rapporte pas d'erreur (mais assignera les codes d'erreurs) ;
- **PDO::ERRMODE_WARNING** - émet un warning ;
- **PDO::ERRMODE_EXCEPTION** - lance une exception.


Trouver les chaines de connexion peut parfois être légèrement difficile. Si les détails sur les DSN dans le manuel de PHP ne vous aide pas, je vous conseille l'excellente contribution de Guillaume Rossolini portant sur les chaines de connexion (DSN) PDO :
http://php.developpez.com/faq/?page=pdo#pdo-connect

Voilà, vous devriez maintenant être en mesure de vous connecter !
Pour la suite du document, nous considérons que la connexion PDO fut précédemment établie dans la variable $pdo, tel que dans l'exemple ci-dessus.

III.b - Sans les requêtes préparées

Si vraiment vous ne voulez pas utiliser les requêtes préparées, vous pouvez utiliser les méthodes query() et exec().

La différence entre ces deux méthodes est que query() retournera un jeu de résultats sous la forme d'un objet PDOStatement, alors que exec() retournera uniquement le nombre de lignes affectées. En d'autres termes, vous utiliserez query() pour des requêtes de sélection (SELECT) et exec() pour des requêtes d'insertion (INSERT), de modification (UPDATE) ou de suppression (DELETE).

Exemple - Effectuer une query et un fetch :	Exemple - Effectuer une query et un fetchAll :	Exemple - Effectuer un exec :
PDO $query = 'SELE $arr = $pdo- >query($query)	mysql_ $query = 'SELHERE*bERM;foo WHERE bar=1;'; $result = mysql_query($query); $arr = mysql_fetch_assoc($result);	
PDO $query = 'SELE $stmt = $pdo- >query($query) $arrAll = $stm >fetchAll();	mysql_ $query = 'SELHERE*bERM0foo WHERE bar<10;'; $result = mysql_query($query); $arrAll = array(); while($arr = mysql_fetch_assoc($result)) ignes $arrAll[] = $arr;	
PDO $query = 'DELE	mysql_ $query = 'DELREbERM0foo WHERE bar<10;'; mysql_query($query);	

PDO	mysql_
`$rowCount = $p >exec($query);`	`$rowCount = mysql_affected_rows();`

Petite mise en garde ; un fetchAll() chargera toutes les données en mémoire d'un seul coup. Évidemment, si la quantité de données est particulièrement importante, vous pourriez éprouver des problèmes de performance.
Lorsque c'est possible, il est donc préférable de traiter une seule ligne de résultat à la fois.
Quoi qu'il en soit, dans plusieurs cas, la méthode fetchAll() demeure un raccourci très pratique !

Eh bien, PDO vous fait un peu moins peur maintenant! Ce n'est finalement pas plus dur que votre façon habituelle de travailler n'est-ce pas ?

III.c - Avec les requêtes préparées

Nous allons maintenant voir comment effectuer des requêtes préparées avec PDO, comparativement à mysqli_ et mysql_.

Pour les requêtes préparées, si le SGBD que vous utilisez ne supporte par ce type de requêtes, PDO s'occupera d'émuler cette fonctionnalité. La requête sera construite et exécutée comme une requête normale au moment du execute().

Exemple - Effectuer un prepare et un fetchAll:

PDO
```
//Préparer la requête
$query = 'SELECT *'
 . ' FROM foo'
 . ' WHERE id=?'
  . ' AND cat=?'
 . ' LIMIT ?;';
$prep = $pdo-
>prepare($query);

//Associer des valeurs aux pla
$prep-
>bindValue(1, 120, PDO::PARAM_
$prep-
>bindValue(2, 'bar', PDO::PARA
$prep-
>bindValue(3, 10, PDO::PARAM_I

//Compiler et exécuter la requ
$prep->execute();

//Récupérer toutes les données
$arrAll = $prep-
>fetchAll();

//Clore la requête préparée
$prep->closeCursor();
$prep = NULL;
```

mysqli_
```
//Préparer la requête
$query = 'SELECT *'
 . ' FROM foo'
 . ' WHERE id=?'
  . ' AND cat=?'
 . ' LIMIT ?;';
$prep = $mysqli-
>prepare($query);

//Associer des valeurs aux pla
$id = 120;
$cat = 'bar';
$limit = 1;
$prep-
>bind_param('isi', $id, $cat,
$prep-
>bind_result($col1, $col2);

//Compiler et exécuter la requ
$prep->execute();

//Récupérer toutes les données
$arrAll = array();
while($prep->fetch())
   $arrAll[] = array(
  'col1' => $col1,
  'col2' => $col2
  );

//Clore la requête préparée
$prep->close();
```

mysql_
```
//Préparer la requête
$query = 'PREPARE stmt_name'
 . ' FROM "SELECT *'
  . ' FROM foo'
  . ' WHERE id=?'
   . ' AND cat=?'
  . ' LIMIT ?";';
mysql_query($query);

//Associer des valeurs aux place holders
$query = 'SET @paramId = 120;';
mysql_query($query);
$query = 'SET @paramCat = "bar";';
mysql_query($query);
$query = 'SET @paramLimit = 10;';
mysql_query($query);

//Compiler et exécuter la requête
$query = 'EXECUTE stmt_name'
 . ' USING @paramId,'
  . ' @paramCat,'
  . ' @paramLimit;'
$result = mysql_query($query);

//Récupérer toutes les données retournées
$arrAll = array();
while($arr = mysql_fetch_assoc($result))
     $arrAll[] = $arr;

//Clore la requête préparée
$query = 'DEALLOCATE PREPARE stmt_name;';
mysql_query($query);
```

III.d - Réutiliser une requête préparée pour gagner en performance

Outre le fait que vos paramètres sont bien protégés, l'avantage initial des requêtes préparées est la réutilisation du moule de la requête. En effet, le SGBD à déjà effectué une partie du traitement sur la requête. Il est donc possible

de ré-exécuter la requête avec de nouvelles valeurs, sans pour autant devoir reprendre le traitement du départ; le découpage et l'interprétation ont déjà été fait !

Réutiliser une requête préparée
```php
$query = 'INSERT INTO foo (nom, prix) VALUES (?, ?);';
$prep = $pdo->prepare($query);

$prep->bindValue(1, 'item 1', PDO::PARAM_STR);
$prep->bindValue(2, 12.99, PDO::PARAM_FLOAT);
$prep->execute();

$prep->bindValue(1, 'item 2', PDO::PARAM_STR);
$prep->bindValue(2, 7.99, PDO::PARAM_FLOAT);
$prep->execute();

$prep->bindValue(1, 'item 3', PDO::PARAM_STR);
$prep->bindValue(2, 17.94, PDO::PARAM_FLOAT);
$prep->execute();

$prep = NULL;
```

Dans ce type de cas de figure qui oblige souvent l'exécution de requêtes dans des boucles, les requêtes préparées représentent une optimisation à ne pas négliger.

IV - Les place holders

IV.a - Nommage des place holders

Un autre avantage de PDO est qu'il permet l'utilisation de *place holders* nommés, c'est-à-dire d'attribuer un nom au *place holder*, plutôt que d'utiliser des points d'interrogation et un indicateur de position numérique.

Comment utiliser les place holders
```php
$query = 'SELECT * FROM foo WHERE id=:id AND cat=:categorie LIMIT :limit;';
$prep = $pdo->prepare($query);

$prep->bindValue(':limit', 10, PDO::PARAM_INT);
$prep->bindValue(':id', 120, PDO::PARAM_INT);
$prep->bindValue(':categorie', 'bar', PDO::PARAM_STR);
$prep->execute();

$arrAll = $prep->fetchAll();

$prep->closeCursor();
$prep = NULL;
```

L'ordre de d'appel des méthodes bindValue() n'a donc plus d'importance.

IV.b - Comment faire face à un nombre de place holders dynamique ?

Dans certaines circonstances, le nombre de *place holders* doit être dynamique. Malheureusement, il n'existe pas de façon élégante de procéder. L'exemple suivant illustre bien la problématique et de quelle façon s'y prendre :

Quantité de place holders variable
```php
$arr = array(12, 62, 61, 36, 92); //Le nombre d'éléments est variable.

//Créer une chaine de place holder
$arrPH = array();
foreach($arr as $elem)
    $arrPH[] = '?';
$strPH = implode(',', $arrPH); //Contient: ?,?,?,?,?
```

Quantité de place holders variable
```
//Préparer la requête
$query = 'SELECT * FROM foo WHERE id IN(' . $strPH . ');';
$prep = $pdo->prepare($query);

//Associer les valeurs aux place holders
for($i=0;$i<count($arr);$i++)
    $prep->bindValue($i+1, $arr[$i], PDO::PARAM_INT);

$prep->execute();
$arrAll = $prep->fetchAll();
$prep->closeCursor();
$prep = NULL;
```

... Eh non, la perfection n'est toujours pas de ce monde !

V - Autres cas particuliers

V.a - Trouver le nombre de lignes retournées

Il existe plusieurs cas de figure où il est utile de calculer le nombre de lignes retournées dans un jeu de résultats. Traditionnellement, cela était fait avec la fonction mysql_num_rows() pour mysql_ ou mysqli_stmt::num_rows() pour mysqli. PDO suit la même logique que MySQLi, en ce sens que vous pouvez accéder au compte des lignes via la méthode PDOStatement->rowCount() du statement.

Utilisation de PDOStatement->rowCount()
```
$query = 'SELECT * FROM foo;';
$pdo->prepare($query);
$prep = $pdo->execute();

echo $prep->rowCount() . ' résultat(s)';

//Autre manipulations hors-contexte
$arrAll = $prep->fetchAll();
var_dump($arrAll);
```

Même si vous n'utilisez pas de requêtes préparées et que vous optez pour la méthode query(), celle-ci retourne tout de même un statement PDO, vous avez donc accès à toutes les méthodes de la classe PDOStatement sur l'objet retourné par la méthode query().

Utilisation de PDOStatement->rowCount()
```
$query = 'SELECT * FROM foo;';
$statement = $pdo->query();

echo $statement->rowCount() . ' résultat(s)';

//Autre manipulations hors-contexte
$arrAll = $statement->fetchAll();
var_dump($arrAll);
```

Il est aussi intéressant de noter que vous ne devriez plus avoir besoin d'utiliser une fonction (ou méthode) de calcul du nombre de lignes pour déterminer si le jeu de résultats est vide ou pas. En effet, si aucun résultat n'est retourné, la méthode fetch() retournera FALSE et la méthode fetchAll() retournera un tableau vide. Je prends la peine de le mentionner car *tous les SGBD n'ont pas nécessairement le même comportement en ce qui concerne le calcul du nombre de lignes retournées*. Autant que possible, il est préférable d'éviter (encore une fois, si possible) d'utiliser ce type d'appels.

Éviter de calculer inutilement le nombre de lignes retournées via fetch
```
$query = 'SELECT * FROM foo LIMIT 1;';
$pdo->prepare($query);
$prep = $pdo->execute();
```

Éviter de calculer inutilement le nombre de lignes retournées via fetch

```
$arr = $prep->fetch();

if($arr !== false)
{
 echo 'Les données sont disponibles:';
 var_dump($arr);
}
else
{
 echo 'Aucun résultat disponible.';
}
```

Éviter de calculer inutilement le nombre de lignes retournées via fetchAll

```
$query = 'SELECT * FROM foo;';
$pdo->prepare($query);
$prep = $pdo->execute();
$arrAll = $prep->fetchAll();

if(!empty($arrAll))
{
 echo 'Les données sont disponibles:';
 foreach($arrAll as $arr)
 {
  echo '<br />Une ligne:';
  var_dump($arr);
 }
}
else
{
 echo 'Aucun résultat disponible.';
}
```

V.b - Trouver le nombre de lignes affectées

Si vous avez besoin de connaitre le nombre de lignes affectées par une requête INSERT, UPDATE ou DELETE, il vous suffit de savoir que ce résultat est transmis comme valeur de retour de la méthode PDOStatement->rowCount() dans le cas où vous avez utilisé une requête préparée.

Nombre de lignes affectées par une requête préparée

```
$query = 'DELETE FROM foo WHERE bar=?;';
$pdo->prepare($query);
$pdo->bindValue(1, 6, PDO::PARAM_INT);
$prep = $pdo->execute();

echo $prep->rowCount() . ' ligne(s) affectée(s).';
```

Si vous détestez toujours autant les requêtes préparées, voici comment trouver ce même résultat via une exécution directe :

Nombre de lignes affectées par une exécution directe

```
$query = 'DELETE FROM foo WHERE bar=6;';
$nbr = $pdo->exec($query);

echo $nbr() . ' ligne(s) affectée(s).';
```

V.c - Explication du faux bogue de la clause LIMIT

À leurs débuts avec PDO, beaucoup de gens -- moi inclus -- ont cru qu'il y avait un bogue lorsque l'on utilise un *place holder* dans une clause LIMIT ; pourtant, il s'agit bien d'une valeur, alors ça devrait fonctionner !

Par défaut, si le 3e paramètre de la méthode bindValue() est ignoré, le type utilisé sera PDO::PARAM_STR. Ce qui signifie que les valeurs auront des guillemets de part et d'autre pour les délimiteurs. En général, ça ne pose pas problème, mais dans le cas de la clause LIMIT, c'est problématique :

Code générant une requête invalide
```
$query = 'DELETE FROM foo LIMIT ?;';
$prep = $pdo->prepare($query);

$prep->bindValue(1, 10);  //Le problème est donc à cette ligne
$prep->execute();
```

Comme aucun type n'est spécifié, PDO gère la valeur 10 comme une chaine de caractères. La requête réellement exécutée est donc :

Requête invalide générée par le code ci-haut
```
DELETE FROM foo LIMIT '1';
```

Les guillemets de part et d'autre de la valeur 1 provoquent donc l'erreur suivante :

« ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near "1" at line 1 »
Voilà, maintenant vous ne pourrez plus dire que PDO c'est moche à cause du non support des clauses LIMIT. Vous savez maintenant qu'il suffit de définir le 3e paramètre de la méthode bindValue() pour PDO::PARAM_INT !

Code générant une requête valide
```
$query = 'DELETE FROM foo LIMIT ?;';
$prep = $pdo->prepare($query);

$prep->bindValue(1, 10, PDO::PARAM_INT);
$prep->execute();
```

VI - Conclusion

Si les requêtes préparées ont été introduites avec MySQL 4.1, il faut admettre que leur utilisation était plutôt complexe, longue et surtout pénible. MySQLi a permis de rendre la chose plus accessible, mais somme toute assez complexe. Si certains préfèreront la syntaxe de PDO, l'avantage premier de ce dernier n'est pas pour autant de faire des requêtes préparées en MySQL, mais bien de permettre d'utiliser une syntaxe uniforme dans l'utilisation des fonctions et méthodes d'accès.

N'oubliez pas ; lors de l'utilisation des requêtes préparées, vous ne pouvez associer que des valeurs et non des commandes SQL.

Remerciements à jacques_jean pour ses relectures.