# Les bonnes pratiques de la classe Db sur Prestashop 1.4

## Les bonnes pratiques de la classe Db sur Prestashop 1.4

*Cet article a été écrit par Raphaël Malié, et publié sur le blog de PrestaShop le 8 août 2011.*

### Préambule

La grande majorité des modules et des développements à effectuer sur PrestaShop nécessitent d'utiliser ou d'insérer des informations dans la base de données. L'utilisation de la classe coeur DB est donc une étape obligatoire pour tout développeur. En plus de fournir une abstraction potentielle pour d'autres types de bases de données relationnelles, la classe DB fournit plusieurs outils destinés à simplifier la vie !

Cet article a pour but d'expliquer ses différentes méthodes, dans quel contexte les utiliser et les bonnes pratiques à avoir en développement.
Les fondements de la classe

La classe DB est en réalité constituée de deux classes :

- La classe DB dans le fichier ~/classes/Db.php qui est abstraite
- La classe MySQL dans le fichier ~/classes/MySQL.php qui étend DB

DB est un pseudo singleton, elle peut tout de même être instanciée au besoin manuellement car le constructeur est public, mais au sein de PrestaShop il est demandé d'accéder à l'instance comme ceci :

```
$db = Db::getInstance();
```

Dans certains cas vous croiserez des appels comme ceci dans le code :

```
$db = Db::getInstance(_PS_USE_SQL_SLAVE_);
```

Si l'utilisateur de PrestaShop permet l'utilisation de serveurs MySQL esclaves dans son architecture, alors la connexion de cette dernière instance pourra se faire sur les serveurs esclaves. Il ne faut utiliser la constante *PS_USE_SQL_SLAVE* en argument que dans le cadre de requêtes en lecture (SELECT, SHOW, etc..), et uniquement si celles-ci n'ont pas besoin d'avoir un résultat mis à jour immédiatement. Si vous faites une requête select juste après une insertion sur la même table, il est nécessaire de la faire sur le serveur maître.
Les différentes méthodes

### La méthode autoExecute()

Cette méthode permet de générer automatiquement l'insertion ou la mise à jour de la base à partir d'un tableau de données. Il faut utiliser cette méthode à la place de faire des requêtes INSERT ou UPDATE, sauf si ces requêtes sont un peu complexes (utilisation de fonctions SQL, requêtes imbriquées, etc.). L'avantage de tout faire via une seule méthode est de centraliser les appels. Le jour où il y a un traitement en particulier à faire sur certaines tables lors de l'insertion il est ainsi possible de le faire en surchargeant cette méthode via le système d'override de la 1.4 de PrestaShop.

**Exemple fictif :**

```
$target = Tools::getValue('id');
$name = Tools::getValue('name');
Db::getInstance()->autoExecute('target_table', array(
        'id_target' =>    (int)$target,
        'name' =>    pSQL($name),
), 'INSERT');
```

L'appel à cette méthode produit la requête SQL suivante :

```
INSERT INTO `target_table` (`id_target`, `name`) VALUES (10, 'myName')
```

**Important :**

- Veillez à toujours protéger vos données avant de les passer à autoExecute()
- Dans l'exemple, on s'assure que id_target soit un entier et que name soit protégé contre les injections SQL avec pSQL()
- Dans le cadre de PrestaShop, le nom des tables doit toujours être précédé du préfixe utilisé, contenu dans la constante *DB_PREFIX*
- En remplaçant le troisième argument par UPDATE il est possible de générer à la place une requête UPDATE. Dans ce cas-là, il est possible de passer des restrictions SQL (par exemple : …->autoExecute('table', $data, 'UPDATE', 'myField = 13 AND id < 8');

## La méthode autoExecuteWithNullValues()

Cette méthode fait la même chose que autoExecute(), a une subtilité près : les chaînes vides et valeurs nulles sont remplacées par des NULL SQL. Cette méthode est donc à utiliser si vos champs acceptent les valeurs nulles, afin qu'ils valent NULL plutôt qu'une chaîne vide.

Cette méthode est en particulier utile lorsque vous souhaitez faire une insertion vide dans une table avec auto increment. En effet, la clé primaire en auto increment sera alors inséreé avec une valeur NULL évitant ainsi une requête d'insertion sans aucun champ.

## La méthode delete($table, $where = false, $limit = false, $use_cache = 1)

Cette méthode est l'équivalent de autoExecute() en version DELETE. Elle est à utiliser pour les mêmes raisons. L'argument $limit permet de limiter le nombre d'enregistrements que l'on souhaite supprimer. L'autre avantage d'utiliser cette méthode est qu'elle sera utilisée par le système de cache des requêtes SQL de PrestaShop et effacera donc les requêtes concernées en cache sauf si l' argument $use_cache vaut false.

Exemple :

```
Db::getInstance()->delete('target_table', 'myField < 15', 3);
```

va générer la requête

```
DELETE FROM target_table WHERE myField < 15 LIMIT 3
```

## La méthode Execute($sql, $use_cache = 1)

Cette méthode exécute la requête SQL donnée. Elle n'est à utiliser que pour les requêtes en écriture (INSERT, UPDATE, DELETE, TRUNCATE…) car elle supprime de plus le cache de requêtes (sauf si l' argument $use_cache vaut false).

Exemple :

```
$sql = 'DELETE FROM '._DB_PREFIX_.'product WHERE date_upd < NOW()';
if (!Db::getInstance()->Execute($sql))
        die('Erreur etc.)';
```

### La méthode ExecuteS($sql, $array = true, $use_cache = 1)

Cette méthode exécute la requête SQL donnée et **charge l'ensemble des résultats qu'elle retourne dans un tableau multidimensionnel**. Elle n'est à utiliser que pour les requêtes en **lecture** (SELECT, SHOW, etc.). Les résultats de cette requête seront mis en cache, sauf si l'argument $use_cache vaut false. Le second argument $array est déprécié et ne doit plus être utilisé, laissez-le à true.

Exemple :

```
$sql = 'SELECT * FROM '._DB_PREFIX_.'shop';
if ($results = Db::getInstance()->ExecuteS($sql))
        foreach ($results as $row)
                echo $row['id_shop'].' :: '.$row['name'].'<br />';
```

### La méthode getRow($sql, $use_cache = 1)

Cette méthode exécute la requête SQL donnée et **récupère la première ligne de résultats**. Elle n'est à utiliser que pour les **requêtes en lecture** (SELECT, SHOW, etc.). Les résultats de cette requête seront mis en cache, sauf si l'argument $use_cache vaut false.

Attention : cette méthode ajoute automatiquement une clause LIMIT à la requête. Veillez à ne pas en ajouter une vous-même manuellement.

Exemple :

```
$sql = 'SELECT * FROM '._DB_PREFIX_.'shop
        WHERE id_shop = 42';
if ($row = Db::getInstance()->getRow($sql))
        echo $row['id_shop'].' :: '.$row['name'];
```

### La méthode getValue($sql, $use_cache = 1)

Cette méthode exécute la requête SQL donnée et **récupère uniquement le premier résultat de la première ligne**. Elle n'est à utiliser que pour les **requêtes en lecture** (SELECT, SHOW, etc.). Les résultats de cette requête seront mis en cache, sauf si l'argument $use_cache vaut false.

Attention : cette méthode ajoute automatiquement une clause LIMIT à la requête. Veillez à ne pas en ajouter une vous-même manuellement.

Exemple :

```
$sql = 'SELECT COUNT(*) FROM '._DB_PREFIX_.'shop';
   $totalShop = Db::getInstance()->getValue($sql);
```

### La méthode NumRows()

Cette méthode met en cache et retourne **le nombre de résultats de la dernière requête SQL**.

**Attention** : cette méthode n’est pas dépréciée mais il est fortement déconseillé de l’utiliser pour des raisons de bonnes pratiques. En effet, il vaut mieux récupérer le nombre de résultats via une requête de type SELECT COUNT( * ) au préalable.

## Quelques méthodes annexes

- Insert_ID() : retourne l’identifiant créé de la dernière requête INSERT exécutée
- Affected_Rows() : retourne le nombre de lignes concernées par la dernière requête UPDATE ou DELETE exécutée
- getMsgError() : retourne le dernier message d’erreur si une requête a échoué
- getNumberError() : retourne le dernier numéro d’erreur si une requête a échoué