Db class good practices for Prestashop 1.4

Db class good practices Prestashop 1.4

This article was written by Raphaël Malié, and first published on the PrestaShop blog, on August 8th, 2011.

Introduction

Most modules and developments on PrestaShop require you to use or enter information on a database. Any developer sees the core DB class as an essential part of the process. On top of providing potential abstraction for other types of relational database, the DB class has several tools to make life simpler!

Learn about the various methods, when to use them and the best practices for the development stage.

Class basics

The DB class is made up of two classes:

- DB class in the file `~/classes/Db.php` is abstract
- MySQL class in the file `~/classes/MySQL.php` builds on DB

Although DB is a pseudo-singleton, it can still be activated manually if necessary as the developer is public. However, in PrestaShop it must be accessed as follows:

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

In some cases you may see requests as per the following code:

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

When the above is connected, it could enter slave servers if the PrestaShop user allows the use of slave MySQL servers in their architecture. The standard *PS_USE_SQL_SLAVE* argument should only be used for read-only queries (`SELECT`, `SHOW`, etc.), and only if a result does not need to be immediately updated. It is necessary you use the master server to make a select query just after entering something on the same table.

Different methods

The autoExecute() method

This method automatically generates the insertion or update of the base from a table of data. This method should be used instead of making `INSERT` or `UPDATE` requests unless these requests are slightly complex (using SQL functions, intersect queries etc.). The benefit of using one method to do everything is to centralize requests. You can edit this method using PrestaShop's 1.4 override system when there is a particular process to apply to tables during insertion..

Dummy example:

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

Requesting this method results in the following SQL query:

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

Important :

- Always ensure that your data is protected before transferring them to `autoExecute()`
- In the example, the id_target must be an integer and the name must be protected against SQL injections with `pSQL()`
- With PrestaShop, table names must always be preceded with the prefix, included in the constant *DB_PREFIX*
- You can generate an UPDATE query by replacing the third argument with `UPDATE`. In this case, you can bypass SQL restrictions (for example: `…->autoExecute('table', $data, 'UPDATE', 'myField = 13 AND id < 8');`).

The autoExecuteWithNullValues() method

This method does the same as `autoExecute()` but with one subtle difference: empty strings and null values are replaced by SQL NULLs. This method can be used if your fields accept null values and results in them being NULL rather than an empty string.

This method is particularly useful when using auto increment to enter an empty entry on a table. The unique key in auto increment will then be entered as NULL thus avoiding an entry query with no field.

Delete method ($table, $where = false, $limit = false, $use_cache = 1)

This method is the DELETE version of `autoExecute()`. It can be used for the same purpose. The $limit argument limits the number of saved items you can delete. The other benefit of this method is that it can be used with PrestaShop's SQL query cache system and deletes the cached queries unless the $use_cache argument is false.

Example :

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

will generate the following query:

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

The execute($sql, $use_cache = 1) method

This method executes the given SQL query. It should only be used for write-only queries (INSERT, UPDATE, DELETE, TRUNCATE etc.) as it also deletes the query cache (unless the $use_cache argument is false).

Example :

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

The executeS($sql, $array = true, $use_cache = 1) method

This method executes the given SQL query and loads all the results on a multidimensional table. It should not be used with read-only queries (SELECT, SHOW etc.). The query results will be cached unless the argument $use_cache is false. The second argument $array is depreciated and should not be used, leave it as 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 />';
```

The getRow($sql, $use_cache = 1) method

This method executes the given SQL query and collects the first line of results. It should only be used with read-only queries (SELECT, SHOW etc.). The query results will be cached unless the argument $use_cache is false.

Warning: this method automatically adds a LIMIT clause to the query. Ensure that you do not add one manually.

Example :

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

The getValue($sql, $use_cache = 1) method

This method executes the given SQL query and only collects the first result on the first line. It should only be used with read-only queries (SELECT, SHOW etc.). The query results will be cached unless the argument $use_cache is false.

Warning: this method automatically adds a LIMIT clause to the query. Ensure that you do not add one manually.

Example :

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

The NumRows() method

This method caches and displays the number of results from the last SQL query.

Warning: this method is not depreciated but we strongly advise you not to use it for reasons of best practice. It is actually better to collect the number of results via a `SELECT COUNT( * )` query beforehand.

Some other methods

- `Insert_ID()`: displays the ID created by the last executed INSERT query
- `Affected_Rows()`: displays the number of lines affected by the last executed UPDATE or DELETE query
- `getMsgError()`: displays the last error message if a query has failed
- `getNumberError()`: displays the last error number if a query has failed