Best Practices of the Db Class -

Table of content

Best Practices of the Db Class

Most of the time, creating a module or overriding PrestaShop means using or inserting data in the database. Knowing how to properly use the DB core class is therefore mandatory for developers. Besides providing you with an abstraction for other potential database system, the DB class offers several tools to make your life easier.

This page explains the various methods, the contexts in which they should be used, and the development best practices.

At the bottom of the page are the main differences in the DB class between version 1.4 and 1.5+ of PrestaShop.

Fundamentals

The DB class is really made of two classes:

- The `Db` class, which can found in the `/classes/db/Db.php`, and is abstracted.
- A subclass which extends the `Db` class. Currently, three class abstractions are supported as subclasses: MySQL, MySQLi and PDO.
  PDO is used by default; however, if the PDO extension is not installed on the server, the MySQLi extension is used instead. And if MySQLi is not installed either, then MySQL is used.

Db is a pseudo-singleton, as it can still be manually instantiated, because its constructor is public. However, within PrestaShop, it is recommended to instantiate it this way:

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

In some cases, you might encounter this alternative:

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

If PrestaShop's database user allows the use of MySQL slave servers in its architecture, then this last instance's connection can be done on the slave servers.
You should only use the *PS_USE_SQL_SLAVE* argument when making read-only queries (`SELECT`, `SHO` , etc.), and only if these do not need a result to be immediately updated with a result. If you make a query on a table right after inserting data in that same table, you should make that query on the master server.

The available methods

insert()

Method signature: `insert($table, $data, $null_values = false, $use_cache = true, $type = Db::INSERT, $add_prefix = true)`.

This method was created to automatically generate data insertion in the database, from a data table. It should be used instead of doing `INSERT` queries, unless these queries are rather complex (use of SQL functions, nested queries, etc.).

Building every query using one method allows you to centralize your calls. If one day you need to perform a specific processing on some tables during data insertion, you can do so by overloading this method using PrestaShop's overriding system.

Fictitious example:

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

Triggering this code generates the following SQL query:

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

Make sure that your data is always checked and protected when doing an insertion. In our example, we want to make sure that we do have an integer with an explicit `(int)` cast, and that the name is protected against SQL injections thanks to the `pSQL()` method.

Method parameters

Parameter	Description
$table	Table's name. The PrestaShop prefix is automatically inserted, you do not have to put it in.
$data	The data array, containing the data to be inserted, with name as keys and data as values.
$null_values	If `true`, then values that are passed as NULL will be inserted as such in the database.
$use_cache	If `false`, PrestaShop's cache management is disabled during this query. Do not change this parameter unless you knew exactly what you are doing.
$type	If you wish to change the insertion, this parameter can take the following constants: `Db::INSERT`, `Db::INSERT_IGNORE` or `Db::REPLACE`.
$add_prefix	If `false`, table prefix will not be automatically added to the table name.

update()

Method signature: `update($table, $data, $where = '', $limit = 0, $null_values = false, $use_cache = true, $add_prefix = true)`

This method works as the `insert()` method does, but for data update (`UPDATE` queries). Both have roughly the same parameters, with `type` gone and these two additions:

Parameter	Description
$where	Takes the update's `WHERE` clause.
$limit	You can limit the number of records that you will update.

⚠ `update()` does not protect your code from hacking attempts (SQL injections, XSS flaws and CRSF breaches). You still have to secure your data yourself.
One PrestaShop-specific securization method is `pSQL($value)`: it helps protect your database against SQL injections.

delete()

Method signature: `delete($table, $where = '', $limit = 0, $use_cache = true, $add_prefix = true)`.

This method is an equivalent to `insert()` and `update()`, only for `DELETE` queries. You should use it for the same reasons.

The `$limit` parameter enables you to limit the number of records to that you wish to delete. The other advantage of this method is that it will be used by PrestaShop's SQL queries cache system, and will therefore delete the affected queries in cache, unles the `$use_cache` is `false`.

Example:

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

...will generate the following query:

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

⚠ `delete()` does not protect your code from hacking attempts (SQL injections, XSS flaws and CRSF breaches). You still have to secure your data yourself.
One PrestaShop-specific securization method is `pSQL($value)`: it helps protect your database against SQL injections.

execute()

Method signature: `execute($sql, $use_cache = 1)`.

This method executes the given SQL query. It should only be used for 'write' queries (`INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`, etc.), because it also deletes the query cache (unles `$use_cache` is set to `fals`).

Example:

```
$sql = 'DELETE FROM '._DB_PREFIX_.'product WHERE active = 0';
if (!Db::getInstance()->execute($sql))
        die('Erreur etc.)';
```

You should use `insert()`, `update()` and `delete()` as much as possible, and only use `execute()` if the query gets too complex.
Please note that this method returns a boolean value (`true` or `false`), not a database resource that can then be used.

`execute()` does not protect your code from hacking attempts (SQL injections, XSS flaws and CRSF breaches). You still have to secure your data yourself.
One PrestaShop-specific securization method is `pSQL($value)`: it helps protect your database against SQL injections.

query()

Method signature: `query($sql)`.

All the method of the DB classes that make SQL query use the `query()` as the common, low-level method. It does the same as the `execute()` method, with two exceptions:

- No cache control management.
- Will not return a boolean; instead returns a database resource that you can use with other DB class methods, such as `nextRow()`.

executeS()

Method signature: `executeS($sql, $array = true, $use_cache = 1)`.

This method executes a given SQL query, and makes that whole resulting data available through a multidimensional array. It should only be used for 'read' queries (`SELECT`, `SHOW`, etc.). The query's results are cached, unless the `$use_cache` parameter is set to `false`. The second parameter, `$array()`, is deprecated and should not be used, leave it as `true`.

Example:

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

`executeS()` does not protect your code from hacking attempts (SQL injections, XSS flaws and CRSF breaches). You still have to secure your data yourself.
One PrestaShop-specific securization method is `pSQL($value)`: it helps protect your database against SQL injections.

getRow()

Method signature: `getRow($sql, $use_cache = 1)`.

This method executes a given SQL query and retrieves the first row of results. It should only be used with 'read' queries (`SELECT`, `SHOW`, etc.). The query's results are cached, unless the `$use_cache` parameter is set to `false`.

> This method automatically adds a LIMIT clause to the query. Be careful not to 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'];
```

> `getRow()` does not protect your code from hacking attempts (SQL injections, XSS flaws and CRSF breaches). You still have to secure your data yourself.
> One PrestaShop-specific securization method is `pSQL($value)`: it helps protect your database against SQL injections.

getValue()

Method signature: `getValue($sql, $use_cache = 1)`.

This method executes a given SQL query and retrieves the first value of the first row of results. It should only be used with 'read' queries (`SELECT`, `SHOW`, etc.). The query's results are cached, unless the `$use_cache` parameter is set to `false`.

> This method automatically adds a LIMIT clause to the query. Be careful not to add one manually.

Example:

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

> `getValue()` does not protect your code from hacking attempts (SQL injections, XSS flaws and CRSF breaches). You still have to secure your data yourself.
> One PrestaShop-specific securization method is `pSQL($value)`: it helps protect your database against SQL injections.

NumRows()

This method caches and returns the number of results from the most recent SQL query;

> This method has not yet been deprecated, but it is still not recommended to use for best-practices reasons. Indeed, it is better to retrieve the number of results using a `SELECT COUNT(*)` before.

A few other methods

- `Insert_ID()`: returns the ID created during the latest `INSERT` query.
- `Affected_Rows()`: returns the number of lines impacted by the latest `UPDATE` or `DELETE` query.
- `getMsgError()`: returns the latest error message, if the query has failed.
- `getNumberError()`: returns the latest error number, if the query has failed.

Security

Note that none of the above methods escape the query itself. You will have to do that using either `pSQL()` or `bqSQL()`.

`pSQL()` is an alias for `Db::getInstance()->escape($string, $htmlOK);`

It has the following PHPDoc comment:

```
/**
 * Sanitize data which will be injected into SQL query
 *
 * @param string $string SQL data which will be injected into SQL query
 * @param bool $htmlOK Does data contain HTML code ? (optional)
 * @return string Sanitized data
 */
```

It accepts a string that will be sanitized by the function. If your string contains HTML-code, be sure to pass the argument `$htmlOK = true` as well.

`bqSQL()` can also be used. Note that besides escaping the ` character, it also calls `pSQL()` afterwards, but without the option to sanitize HTML.