Accessing the database

Accessing the database

The database structure

By default, PrestaShop's database tables start with the `ps_` prefix. This can be customized during installation

All table names are in lowercase, and words are separated with an underscore character ("_"):

- `ps_employee`
- `ps_manufacturer`
- `ps_product`
- `ps_product_comment`
- `ps_shop_url`

When a table establishes the links between two entities, the names of both entities are mentioned in the table's name. For instance, `ps_category_product` links products to their category.

A few details to note about tables:

- Tables which contain translations must end with the `_lang` suffix. For instance, `ps_product_lang` contains all the translations for the `ps_product` table.
- Tables which contain the records linking to a specific shop must end with the `_shop` suffix. For instance, `ps_category_shop` contains the position of each category depending on the store.

There is also a couple of standard practices for data rows within a table:

- Use the `id_lang` field to store the language associated with a record.
- Use the `id_shop` field to store the store associated with a record.

The DBQuery class

The DBQuery class is a query builder which helps you create SQL queries. For instance:

```
$sql = new DbQuery();
$sql->select('*');
$sql->from('cms', 'c');
$sql->innerJoin('cms_lang', 'l', 'c.id_cms = l.id_cms AND l.id_lang = '.(int)$id_lang);
$sql->where('c.active = 1');
$sql->orderBy('position');
return Db::getInstance()->executeS($sql);
```

Here are some of the methods from this class:

Method name and parameters	Description
__toString()	Generate and get the query.
build()	Generate and get the query (return a string).
from(string $table, mixed $alias = null)	Set table for FROM clause.
groupBy(string $fields)	Add a GROUP BY restriction.
having(string $restriction)	Add a restriction in the HAVING clause (each restriction will be separated by an AND statement).
innerJoin(string $table, string $alias = null, string $on = null)	Add a INNER JOIN clause, E.g. $this->innerJoin('product p ON ...').

join(string $join)	Add a JOIN clause, E.g. $this->join('RIGHT JOIN'. _DB_PREFIX_.'product p ON ...');.
leftJoin(string $table, string $alias = null, string $on = null)	Add a LEFT JOIN clause.
leftOuterJoin(string $table, string $alias = null, string $on = null)	Add a LEFT OUTER JOIN clause.
limit(string $limit, mixed $offset = 0)	Limit results in query.
naturalJoin(string $table, string $alias = null)	Add a NATURAL JOIN clause.
orderBy(string $fields)	Add an ORDER B restriction.
select(string $fields)	Add fields in query selection.
where(string $restriction)	Add a restriction in WHERE clause (each restriction will be separated by an AND statement).

The ObjectModel class

When needing to dive deep, you have to use the ObjectModel class. This is the main object of PrestaShop's object model. It can be overridden... with precaution.

It is an Active Record kind of class (see: http://en.wikipedia.org/wiki/Active_record_pattern). The table attributes or view attributes of PrestaShop's database are encapsulated in this class. Therefore, the class is tied to a database record. After the object has been instantiated, a new record is added to the database. Each object retrieves its data from the database; when an object is updated, the record to which it is tied is updated as well. The class implements accessors for each attribute.

Defining the model

You must use the `$definition` static variable in order to define the model.

For instance:

```
/**
* Example from the CMS model (CMSCore)
*/
public static $definition = array(
  'table' => 'cms',
  'primary' => 'id_cms',
  'multilang' => true,
  'fields' => array(
    'id_cms_category'  => array('type' => self::TYPE_INT, 'validate' => 'isUnsignedInt'),
    'position'         => array('type' => self::TYPE_INT),
    'active'           => array('type' => self::TYPE_BOOL),

    // Language fields
    'meta_description' =>
        array('type' => self::TYPE_STRING, 'lang' => true, 'validate' => 'isGenericName', 'size' => 255),
    'meta_keywords'    =>
        array('type' => self::TYPE_STRING, 'lang' => true, 'validate' => 'isGenericName', 'size' => 255),
    'meta_title'       =>
        array('type' => self::TYPE_STRING, 'lang' => true, 'validate' => 'isGenericName', 'required' => true,
'size' => 128),
    'link_rewrite'     =>
        array('type' => self::TYPE_STRING, 'lang' => true, 'validate' => 'isLinkRewrite', 'required' => true,
'size' => 128),
    'content'          =>
        array('type' => self::TYPE_HTML,   'lang' => true, 'validate' => 'isString', 'size' => 3999999999999),
  ),
);
```

A model for many stores and/or languages

In order to retrieve an object in many languages:

```
'multilang' => true
```

In order to retrieve an object depending on the current store

```
'multishop' => true
```

In order to retrieve an object which depends on the current store, and in many languages:

```
'multilang_shop' => true
```

The main methods

Any overriding of the ObjectModel methods is bound to influence how all the other classes and methods act. Use with care.

Method name and parameters	Description
__construct($id = NULL, $id_lang = NULL)	Build object.
add($autodate = true, $nullValues = false)	Save current object to database (add or update).
associateTo(integer|array $id_shops)	Associate an item to its context.
delete()	Delete current object from database.
deleteImage(mixed $force_delete = false)	Delete images associated with the object.
deleteSelection($selection)	Delete several objects from database.
getFields()	Prepare fields for ObjectModel class (add, update).
getValidationRules($className = _CLASS_)	Return object validation rules (field validity).
save($nullValues = false, $autodate = true)	Save current object to database (add or update).
toggleStatus()	Toggle object's status in database.
update($nullValues = false)	Update current object to database.
validateFields($die = true, $errorReturn = false)	Check for field validity before database interaction.