A Quick and Friendly Introduction to eZPersistentObject

By Thiago Campos Viana
http://thiagocamposviana.blogspot.com/

Index

1 Goal description

In this tutorial we will learn how to use **CRUD features** by extending **eZPersistentObject** class, so it will be possible to **C**reate, **R**ead, **U**pdate, and **D**elete objects in the database in a straightforward way, **without writing SQL** queries.

2 Introduction

Friends are cool, friendly **eZ publish API classes** too, one of those classes is the **eZPersistentObject**, do you know it? No? But you should! You can **avoid** a lot of **headaches** with your data living inside a **database** if you learn how to deal with this class. It is a central class of the eZ publish kernel and is responsible to persist objects to the database without all the complex and boring SQL stuff, by using an **Object Oriented approach** instead. You can view this class as an **abstraction layer between the application and the eZDB** database interface, a mediator that makes possible to create, read, update, and remove data in the database without worrying with low-level database code. **Almost all the classes that store simple data** in the eZ Publish database **extend it**.

You can see the usage of the eZPersistentObject class in:

- Complex datatypes like **eZBinayFile** or **eZStarRating**;
- Most others **kernel central classes** extends this class, like **eZContentObject**, **eZContentClass**, **eZSection**, and **eZRole**;
- Most of the eZ publish kernel modules use classes that extends eZPersistentObject;
- Extensions that has custom database tables should use this class.

When **developing complex extensions** it will become **fundamental** to **understand this class** and its usage(s) so you can use its facilities all the time. In this tutorial we will learn about this class in 3 Steps.

1. Create a sample database table;
2. Create a class that extends eZPersistentObject class to manipulate the table;
3. Test our class doing common database table manipulation tasks.

3 Pre-requisites and target population

This tutorial is intended for eZ publish **intermediate users** who know the basics of extension development. It is recommended to have some knowledge about SQL queries , and database transactions.

4 Step 1: Creating the database table

Let’s start creating a simple database table. In this tutorial we will create a table to store the friendship relations between users. Our database will have only three columns:

user1_id - The id of user 1.

user2_id - The id of the user 2.

status - The status of the friendship.

Every time a user 1 makes a friendship request to a user 2, we store this information in the database. A status 0 means that the request is pending, 1 means that the request was accepted.

Code :

```
CREATE TABLE tutorial_friendship (
  user1_id int(11) NOT NULL,
  user2_id int(11) NOT NULL,
  status int(11) NOT NULL,
  PRIMARY KEY (user1_id, user2_id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Here's the PostgreSQL version.

Code :

```
CREATE TABLE tutorial_friendship (
  user1_id integer DEFAULT 0 NOT NULL,
  user2_id integer DEFAULT 0 NOT NULL,
  status integer DEFAULT 0 NOT NULL,
  PRIMARY KEY (user1_id,user2_id)
);
```

5 Step 2: Creating our class

We start creating the file extension/ezpotutorial/classes/tutorialfriendshipobject.php:
Code :

```
<?php

class TutorialFriendshipObject extends eZPersistentObject
{
     const STATUS_ACCEPTED = 1;
     const STATUS_PENDING = 0;
     /**
     * Construct, use {@link UserExpObject::create()} to create new objects.
     *
     * @param array $row
     */
    protected function __construct(  $row )
    {
        parent::eZPersistentObject( $row );
    }

    public static function definition()
    {
        static $def = array( 'fields' => array(
                             'user1_id' => array(
                                   'name' => 'user1_id',
                                   'datatype' => 'integer',
                                   'default' => 0,
                                   'required' => true ),
                             'user2_id' => array(
                                   'name' => 'user2_id',
                                   'datatype' => 'integer',
                                   'default' => 0,
                                   'required' => true ),
                             'status' => array(
                                   'name' => 'status',
                                   'datatype' => 'integer',
                                   'default' => self::STATUS_PENDING,
                                   'required' => true ),
              ),
              'keys' => array( 'user1_id', 'user2_id' ),
              'class_name' => 'TutorialFriendshipObject',
              'name' => 'tutorial_friendship' );
        return $def;
    }

    public static function create( array $row = array() )
    {
        if( $row['status'] != self::STATUS_ACCEPTED
                and $row['status']!= self::STATUS_PENDING )
        {
            $row['status']= self::STATUS_PENDING;
        }
        $object = new self( $row );
        return $object;
    }


}


?>
```

In this example class, the **constructor receives an associative array** with the name and **values of the object attributes** (user1_id, user2_id, and/or status) and create the eZPersistentObject. The **definition function** returns an associative **array that specify the object metadata** by declaring its fields, keys, database table and so on. When creating a class that inherits from **eZPersistentObject** you need to implement this function. Here's a brief description of all the necessary information:

fields	An associative array of fields which defines which database field (the key) is to fetched and how they map to object member variables (the value). The datatype field can be int, integer, float, double, or string.
keys	An array of fields which is used for uniquely identifying the object in the table.
function_attributes	An associative array of attributes which maps to member functions, used for fetching data with functions.
set_functions	An associative array of attributes which maps to member functions, used for setting data with functions.
increment_key	The field which is incremented on table inserts.
class_name	The classname which is used for instantiating new objects when fetching from the database. The name of the class we are working on.
sort	An associative array which defines the default sorting of lists, the key is the table field while the value is the sorting method which is either **asc** or **desc**. Caution with high volume of data, it can decrease the performance. The sort will be applied to every query when no explicit sort is demanded.
name	The name of the database table

6 Step 3: Testing

First we need create the file extension/ezpotutorial/bin/php/test.php

Code :

```
<?php
set_time_limit ( 0 );
```

```
require 'autoload.php';
$cli = eZCLI::instance();
$script = eZScript::instance( array( 'description' => ( "eZPersistentObject
tutorial.\n\n"),
                                     'use-modules' => true,
                                     'use-extensions' => true) );


$script->startup();
$script->initialize();


// Code Goes Here


$script->shutdown();
?>
```

Then we need to run the following commands in the command line from the root folder of our site:

```
php bin/php/ezcache.php --clear-all --purge
php bin/php/ezpgenerateautoloads.php -e -p
```

To run the script you need to run the following command:

```
php extension/ezpotutorial/bin/php/test.php
```

The explanation of the eZScript API is out of the scope here, for more information there's an article from eZPedia about command line scripts.

6.1 Creating and storing

To create an object, you need to pass an array as parameter, this array contains associative values according to the table field, then to store the object you just need to call the store() function and the object information will be stored in the database table.

Code :

```
$simpleObj = TutorialFriendshipObject::create( array(     'user1_id' => 1,
                                                           'user2_id' => 3


));
$simpleObj->store();
```

Copy and paste the script above in our test.php where you below the "// Code Goes Here", then run your script.

Your database should have a new row:

user1_id	user2_id	status
1	3	0

6.2 Reading objects

Code :

```
$cond = array( 'user1_id' => 1, 'user2_id' => 3);
$simpleObj = eZPersistentObject::fetchObject( TutorialFriendshipObject::definition(),
null, $cond );
$cli->output( $simpleObj->attribute( 'status' ) );
```

The output should be ‘0’.

6.3 Updating

Code :

```
$cond = array( 'user1_id' => 1, 'user2_id' => 3);
$simpleObj = eZPersistentObject::fetchObject( TutorialFriendshipObject::definition(),
null, $cond );
$simpleObj->setAttribute( 'status',1 );
$simpleObj->store();
$cli->output( $simpleObj->attribute( 'status' ) );
```

In this sample we just set the status attribute value to '1' and stored it in the database. If we set the user1_id or user2_id attribute value it will be created a new row in the database table, because that attribute is part of the primary key.

6.4 Deleting

Code :

```
$cond = array( 'user1_id' => 1, 'user2_id' => 3);
eZPersistentObject::removeObject( TutorialFriendshipObject::definition(), $cond );
```

Or:

Code :

```
$cond=array( 'user1_id' => 1, 'user2_id' => 3);

$simpleObj = eZPersistentObject::fetchObject( TutorialFriendshipObject::definition(),
null, $cond );

$simpleObj->remove();
```

See the Appendix for a short explanation of transactions, and how to use them from the eZ Publish API.

6.5 List

Code :

```
// Creates two rows

TutorialFriendshipObject::create( array('user1_id' => 1, 'user2_id' => 3, 'status' =>
self::STATUS_PENDING ))->store();

TutorialFriendshipObject::create( array('user1_id' => 2, 'user2_id' => 4, 'status' =>
self::STATUS_PENDING ))->store();


$cond = array();

$list = eZPersistentObject::fetchObjectList( TutorialFriendshipObject::definition(),
null, $cond );

$cli->output( "Listing objects:\n" );

foreach( $list as $obj )
{

    $cli->output( "Object Status: ". $obj->attribute('status') . "\n" );
}
```

The output should be:

Code :

```
Listing objects:

Object Status: 0

Object Status: 0
```

7 Conclusion

In this tutorial **you learnt** how to use **a fundamental eZ Publish kernel class**, **eZPersistentObject**. This class can be **used to speed up the development of eZ publish extensions** that need **persistent facilities regardless** of which **database is used** and according to a well established design pattern. As most of the kernel classes that are used in extensions inherit from this class, learning how to use this class can help understand how that others classes work.

So from now on, **think twice before starting to write SQL** commands in your extensions. Your homework is to create functions to make the operations described in this tutorial easier and learn how to use sorting and the conditions array.

8 Resources

- eZPersistentObject online documentation.
- eZ publish API Doc
- Complete ezpersistentobject.php code.
- eZPedia Persistent Object Article
- eZPersistentObject Generator Project
- Article from eZPedia about command line scripts
- An Introduction to Developing eZ Publish Extensions.
- UserExp datatype: A sample datatype project that extends eZPersistentObject to store user experience data.
- Creating Datatypes in eZ Publish 4 tutorial.
- Understanding and developing fetch functions tutorial.
- Creating eZ publish objects in php tutorial.

9 Appendix : Transactions, used from eZ Publish's API

Transactions are a fundamental notion of Database Management Systems. I warmly invite you to get acquainted with them, if not already the case, through reading this : http://en.wikipedia.org/wiki/Database_transaction

A typical use case is when an DB row needs to be deleted, while it may be accessed (read, updated) simultaneously by other threads/processes of one application. This does not only apply to high-concurrency types of applications, you should bear this in mind when crafting the DB-access parts of your business logics.

Here is an example of such : removing a row, through the eZPersistentObject API, should be encapsulated in a transaction. Elaborating on the previous DELETE example :

```
// Prime the transaction
$db = eZDB::instance();
$db->begin();

// Do the delete. It will not be actually done until the transaction is committed
$cond = array( 'user1_id' => 1,
               'user2_id' => 3 );
$simpleObj = eZPersistentObject::fetchObject( TutorialFriendshipObject::definition(),
null, $cond );
$simpleObj->remove();

// Commit all changes. The set of changes committed only contains one change here : the
// deletion of a row. A transaction can contain a series of several changes.
$db->commit();
```

The transaction API is much larger than only begin() and commit() presented above. Please refer to lib/ezdb/classes/ezdbinterface.php for a deeper insight.

10 About the author : Thiago Campos Viana

Thiago Campos Viana is a web developer and eZ Publish enthusiast from Brazil.

11 License choice