FaZend: Object Relational Mapping

by Yegor Bugayenko

FaZend is an open-source PHP framework and a continuous integration environment, which simplifies the development and maintenance of your web applications. FaZend is based on Zend Framework. Being one of the most powerful PHP frameworks, Zend Framework is very flexible and abstract. Very often, this flexibility leads to complexity in developed applications. This article explains how FaZend overcomes this drawback and makes the management of persistent data both fast and easy.

REQUIREMENTS

PHP: *5.2+*

Other Software:
- MySQL 5.0+
- FaZend http://code.google.com/p/fazend/
- Zend Framework 1.10+

Related URLs:
- Zend Framework: http://framework.zend.com/
- FaZend: http://www.fazend.com/

What is FaZend? An Environment or a Framework?

Recent studies of the industry indicate that the failure rate of software development projects is growing, and in general, "software quality is not improving but getting worse" [Cerpa09]. Some reports say that up to 85 percent of projects fail to reach their objectives [Jorgensen06]. Despite all efforts, these are sad statistics. One of the most critical factors of project success is complexity of source code. The higher the complexity, the higher the risk of project failure.

One of the best way to decrease source code complexity is to use architecture layers [POSA08, pp.31-51], which isolate components and let us work with their public interfaces. Well-known best practices of layering include functions, classes, and also sets of classes called "libraries" and "frameworks".

Zend Framework is one of the most developed and powerful PHP frameworks available in the market now, and it "provides loose coupling between components" [OPhinney09]. FaZend Framework ("Fully Automated Zend") was designed and developed in order to move forward with decreasing code complexity. This product is just another layer on top of Zend Framework, and it gives web applications a limited set of the most useful components with extremely simplified interfaces.

Besides being an additional abstraction layer on top of Zend Framework, FaZend is a continuous integration environment, which automates building, maintenance, testing and defect tracking of PHP products.

A little background on FaZend: The development of FaZend Framework was started in May 2009. According to **ohloh.com**, there are over 13K lines of code in the framework and over 10K in the server components. The open-source framework is hosted at Google Code SVN repository. To date, there are 38 projects managed by FaZend continuous integration environment. Hardware resources are provided by RackSpace.com, HostGator.com and WebFaction.com. The environment is powered by third-party open-source tools: PHP, Apache, MySQL, phpDocumentor, xdebug, Phing, PHPUnit, PHP_CodeSniffer, PHP_Beautifier, PHP_Depend, Trac, Subversion, CruiseControl and phpUnderControl.

This article is dedicated to one of the core mechanisms inside the open-source FaZend Framework — the object-relational mapping component — which is developed on top of the **Zend_Db** set of classes.

How Zend_Db Manages Persistent Data

Zend Framework gives you a powerful mechanism of database layer abstraction, by means of **Zend_Db_*** classes. There are many articles about its usage, with examples and details available in both the Zend Framework tutorials and in the developers' blogs. In a nutshell, to manage your persistent data with **Zend_Db**, you need to do the following (see Listing 1):

- Configure a default adapter in your **Bootstrap.php**
- Declare classes which extend **Zend_Db_Table**
- Instantiate one of the declared classes
- Execute an SQL query, fetch a row or a "row set"
- Retrieve data

The SQL query could be prepared beforehand as an instance of class **Zend_Db_Select**, which is another abstraction on top of SQL. In Listing 1, the variable `$query` is a good example.

Compared with the direct access to PDO methods, such an approach gives you a number of benefits, including loose coupling and higher maintainability.

In formal terms:

- **Zend_Db_Table** implements "Table Data Gateway" pattern
- **Zend_Db_Table_Row** implements "Row Data Gateway" pattern
- **Zend_Db_Table_Rowset** implements "Record Set" pattern [Fowler08]

However, there is still a lack of "Data Access Object (DAO)" pattern implementation, which would enable true Object-Relational Mapping (ORM). To extend the snippet above with ORM, it would be nice to use such code:

```
$name = $product->person->name;
```

With **Zend_Db**, you can't do this that quickly and simply, but you should manually retrieve data from tables and execute new queries, for example:

```
$personId = $product->person;
$personTable = new Model_Person();
$query = $personTable->select()
  ->where('id = ?', $personId);
$name = $personTable->fetchRow($query)->name;
```

Obviously, the complexity of this code is much higher (remember the failure rate of 85 percent!) than the one-line call above. On the other hand, cohesion of this code is also much weaker, since there is no explicit grouping of the five code lines above. Sooner or later, a new member of your software team may decide to re-factor this code and one of the lines may be lost. If this happens, it will be extremely difficult to understand which one it was and what it contained.

There are a number of proposals being discussed in the Zend Framework wiki about possible ORM implementation on top of `Zend_Db`, however, none of them have been approved by the Zend Framework team so far.

Existing ORM Tools and Frameworks in PHP

There are a few popular frameworks and libraries that enable DAO pattern in PHP, including Doctrine [McNulty09], Propel [Godoy08], Torpor, Qcodo and others. However, none of them are based on `Zend_Db`, and their integration with Zend Framework is rather complex (and will lead to code complexity).

Another big disadvantage of all said libraries is the necessity to duplicate DB schema in PHP code. Sometimes, this process is automated with re-engineering tools that grab DB schema from the server (or from its declaration files in XML, YAML, SQL, etc.) and generate PHP code that declares classes and methods for ORM calls. Needless to say, overall code complexity of the entire product grows extremely fast when you inject auto-generated code into it.

On the other hand, the vast majority of web applications do not have complex DB schemas and don't need most of the instruments provided by the ORM libraries.

How FaZend Makes it Easy and Fast

FaZend Framework offers a simple, fast and easy ORM mechanism without DB schema duplication inside PHP code and without static DB re-engineering. This mechanism is not going to replace all possible persistent data management use cases, but it will be suitable for the majority of situations.

FaZend Framework implements ORM by means of on-fly declaration of DAO classes and discovery of database schemas, i.e.:

```
class Model_Product
  extends FaZend_Db_Table_ActiveRow_product
{
    // ...
}
```

As you see in the snippet above, class `FaZend_Db_Table_ActiveRow_product` is not a real class, but a stub in order to catch this extension call by a dynamic class loader. When you declare class `Model_Product` this way, the PHP class loading mechanism sends a request to FaZend loader, asking it to declare a class. FaZend Framework makes a request

LISTING 1

```
// To configure default adapter
Zend_Db_Table::setDefaultAdapter(
  new Zend_Db_Adapter_Pdo_Mysql(
    array(
      'host' => '127.0.0.1',
      'username' => 'webuser',
      'password' => 'xxxxxxxx',
      'dbname' => 'test',
    )
  )
);

// To declare class
class Model_Person extends Zend_Db_Table
{
  protected $_name = 'person';
  protected $_primary = 'id';
}

// To instantiate the class
$personTable = new Model_Person;

// To fetch a row
$query = $personTable->select()
  ->where('id = ?', 123)
  ->orWhere('name LIKE ?', '%John%')
$person = $personTable->fetchRow($query);

// To retrieve data
if (false !== $person) {
  echo $person->name;
}
```

to the database table `product` and retrieves all information required, by means of a `DESCRIBE `product`` call. Thus, you don't need to declare explicitly the name of the table in the class, as was done in the snippet with `Zend_Db` in Listing 1.

Now, you **implicitly** tell FaZend whether our columns contain plain data values or foreign keys to other rows in other tables. For example, column `product.name` is a plain string value, while `product.person` is a foreign key to some row in `person` table, identified by a primary key `id`. By default, it is assumed that the name of the column is equal to the name of the referenced table. It is also assumed that every table has a primary key named `id`. Having these two key assumptions in mind, you can do:

```
$product = new Model_Product(123);
$person = $product->person;
assert($person instanceof Model_Person);
```

Record set retrieval is also simplified in order to avoid instantiation of table class, i.e.:

```
$products = Model_Product::retrieve()
  ->where('price > ?', 500)
  ->setRowClass('Model_Product')
  ->fetchAll();
foreach ($products as $product) {
  assert($product->person instanceof Model_Person);
}
```

Another instrument is **explicit** class mapping. For example, you have a column **person.dob**, which is of type DATE and contains date of birth of the designated person. This is what you do:

```
FaZend_Db_Table_ActiveRow::addMapping(
  '/^person\.dob$/',
  'Zend_Date'
);
$product = new Model_Product(123);
assert($product->person->dob instanceof Zend_Date);
```

With a simple call to **addMapping()** method, you instruct FaZend Framework to convert everything it gets from **person.dob** column to **Zend_Date**.

A Few Real-Life Examples

Here is a real-life example of a DAO class, with comments for each method. This example is going to give the best explanation of ORM mechanism in FaZend. Consider the DB schema in Listing 2 where MySQL 5.0 is used in the example.

Two entity classes are declared in Listing 3 (class **Model_Person**) and Listing 4 (class **Model_Product**). A simple view is in Listing 5. "Document View" pattern is used in the example, instead of "Model View Controller", for the sake of simplicity [POSA08, p.140]. Further, we explain, step-by-step, every class and method from said Listings.

Model_Person in Listing 3 extends class **FaZend_Db_Table_ActiveRow_person**, and this class is created on-the-fly, using the suffix of the name provided: **person**. This suffix is the exact name of the database table to use. The class implements the "Active Row" design pattern, representing a single row from the table in PHP code scope. Static calls to this class shall remind you of the well-known "Factory" design pattern, since they allow you to create or retrieve instances of the "row" class.

Model_Person::create() is a factory method that creates an instance of the row and returns it. Pay attention that type hinting is used for incoming

LISTING 2

```
CREATE TABLE IF NOT EXISTS `person`
(
  `id` INT NOT NULL AUTO_INCREMENT
    COMMENT "Unique ID of the person",
  `name` VARCHAR(120) NOT NULL COMMENT
    "Full name of the person",
  `dob` DATE COMMENT
    "Date of birth",
  PRIMARY KEY(`id`)
)
AUTO_INCREMENT=1
ENGINE=InnoDB
COMMENT="A human being, potential product owner";

CREATE TABLE IF NOT EXISTS `product`
(
  `id` INT NOT NULL AUTO_INCREMENT
    COMMENT "Unique ID of the product",
  `title` TEXT NOT NULL COMMENT
    "Title of the product",
  `price` INT NOT NULL COMMENT
    "Price of the product, in USD cents",
  `person` INT NOT NULL COMMENT
    "Owner of this product (FK:person.id)",
  PRIMARY KEY(`id`),
  FOREIGN KEY(`person`)
    REFERENCES `person`(`id`)
    ON UPDATE CASCADE
    ON DELETE CASCADE
)
AUTO_INCREMENT=1
ENGINE=InnoDB
COMMENT="A product under development";
```

LISTING 3

```
class Model_Person
  extends FaZend_Db_Table_ActiveRow_person
{
  public static function create($name, Zend_Date $dob)
  {
    $person = new self();
    $person->name = $name;
    $person->dob = $dob->getIso();
    $person->save();
    return $person;
  }

  public function __get($name)
  {
    $method = '_get' . ucfirst($name);
    if (method_exists($this, $method))
      return $this->$method();
    return parent::__get($name);
  }

  protected function _getProducts()
  {
    return Model_Product::retrieveByPerson($this);
  }

  protected function _getAge()
  {
    return Zend_Date::now()->sub($this->dob)->getYear();
  }
}
```

LISTING 4

```
class Model_Product
  extends FaZend_Db_Table_ActiveRow_product
{
  public static function create(
    Model_Person $person,
    FaZend_Bo_Money $price,
    $title)
  {
    $product = new self();
    $product->title = $title;
    $product->price = $price->cents;
    $product->person = $person;
    $product->save();
    return $product;
  }

  public static function retrieveByPerson(
    Model_Person $person)
  {
    return self::retrieve()
      ->where('person = ?', strval($person))
      ->setRowClass('Model_Product')
      ->fetchAll();
  }
}
```

parameters, where possible. It is good practice to make it the responsibility of a caller to validate parameters and do the necessary type casting. Similarly, it is good practice to instantiate "gateway" classes by means of the factory method, instead of their constructor. `Model_Person` is a gateway between persistent data in the database and their PHP users. Instantiation of such a gateway by means of its constructor will lead to a concurrency problem, when two gateways work with the same DB row.

`Model_Person::__get()` is a magic PHP5 method that dispatches property access attempts, hiding internal logic of the class behind an object interface. The method enables two properties in the class, implementing them by means of methods:

```
assert($this->products === $this->_getProducts());
assert($this->age === $this->_getAge());
```

Notice, that `_getAge()` and `_getProducts()` are declared as `protected` methods. This declaration will disallow callers to access them directly.

`Model_Person::_getAge()` calculates the age of the person in years. Since `person.dob` is automatically converted to `Zend_Date`, the calculation of the age is done with just one straight call.

The **class `Model_Product`** class in Listing 4 uses similar declaration notation, but the suffix after `FaZend_Db_Table_ActiveRow_` is changed to `product` in order to instruct FaZend Framework that this class should represent table `product`.

`Model_Product::retrieveByPerson()` is a static factory method that returns a set of rows (instance of class `Zend_Db_Table_Rowset`), where each row is an instance of class `Model_Product`. Method `self::retrieve()` is a generator of a wrapper around two classes: `Zend_Db_Table` and `Zend_Db_Select`. In order to simplify operations with them, the wrapper catches your calls and dispatches them as required. Some of them go to `Zend_Db_Table`, like `fetchAll()`, `fetchRow()`, `delete()`, etc., while others go to `Zend_Db_Select`, like `where()`, `order()`, etc.

Pay attention to the `setRowClass()` call, which instructs the wrapper to do type casting of the retrieved data to the class specified. This is required in PHP 5.2, where the name of the class-caller is not available for static methods. In PHP 5.3, you may skip this explicit instruction, as FaZend Framework will understand who is calling the `retrieve()` method and will perform necessary type casting automatically. However, this call may be useful if you retrieve rows from a `VIEW` rather than a `TABLE` and you need to specify explicit type casting. In such a case, you may even need to change the name of the table you're working with. For example:

```
echo self::retrieve(false)
  ->from('person', array('id', 'name'))
  ->join('product', 'product.person = person.id',
    array('volume' => new Zend_Db_Expr('SUM(product.
price)'))
  ->group('person.id')
  ->order('volume DESC');
```

In this example, `retrieve(false)` means that we should not use the current table in the `FROM` SQL statement, but wait for `from()` to specify the table explicitly, whereby we also can list columns that we need to see in the query result set. They are `id` and `title` in the example above. The statement

LISTING 5

```
1.  $person = new Model_Person((int)$this->param('id'));
2.
3.  <h1>Customer #<?=$person?>:
4.    <?=$this->escape($person->name)?>
5.    (<?=$person->age?> y.o.)</h1>
6.
7.  <p><?=count($person->products)?> product(s)
8.    are in production for the customer:</p>
9.
10. <? foreach ($person->products as $product): ?>
11.   <p>#<?=$product?>:
12.     <?=$this->escape($product->title)?>,
13.     <b><?=$product->price?></b></p>
14. <? endforeach; ?>
```

above will produce the following SQL query (you noticed already that there is no `fetchAll()` or similar call, but rather an `echo()` method on the entire wrapper object, which will go directly to `Zend_Db_Select::__toString()` and will return a string comprising the SQL query):

```
"SELECT id, name, SUM(product.price) AS volume
FROM person
JOIN product ON product.person = person.id
GROUP BY person.id
ORDER BY volume DESC"
```

Here is another example that explains how dynamic binding could be used more extensively than before:

```
return Model_Person::retrieve()
  ->where('dob BETWEEN :start AND :end')
  ->setRowClass('Model_Person')
  ->fetchAll(array(
    'start' => Zend_Date::now()->subYear(25)->getIso(),
    'end' => Zend_Date::now()->subYear(18)->getIso()));
```

Such a dynamic binding is a powerful mechanism when you need to pass many variables into your SQL statement, but the `where()` method cannot accept more than one parameter.

Updates and deletes are performed with the same one-call "fluent interface" syntax. For example, this call will delete all rows in the `product` table which are cheaper than `$price`:

```
Model_Product::retrieve()
  ->where('price < ?', $price)
  ->delete();
```

Another example, below, will rename all **person** records that have any **product** records, converting their names
to upper case:

```
Model_Product::retrieve()
  ->join('product', 'product.person = person.id')
  ->update(array(
    'name' => new Zend_Db_Expr('UPPER(name)')));
```

The **view script** in Listing 5 demonstrates how specified classes may interact and how easily this interaction can be managed. It is important to mention that FaZend Framework will encourage you to avoid direct manipulation with row **id**s. When you will try to access an **id** column of an object, you will get a warning that this is a prohibited manipulation practice. When, and if, you need to get an **id** of the row, just convert the object to **STRING** like it is done in the view script:

```
<?=$person ?>
```

You may have noticed that nowhere in the code was the **id** column used. This is done intentionally in order to encourage programmers to use object-oriented concepts, instead of thinking in terms of rows, columns and tables. When you have an object (**$person** for example), you should forget that this is an interface to a DB row. You should work with it like you would with an object — getting properties from it, setting properties and calling methods. If you're trying to access **id**, it means that you're concerned about the implementation of a storage of this object, and this is an explicit sign of invalid design.

> Such a dynamic binding is a powerful mechanism when you need to pass many variables into your SQL statement.

Cost-Benefit Analysis and Further Plans

There are a number of drawbacks in the proposed ORM mechanism, such as the inability to work with multiple databases at the same time (since the default DB adapter is always used) and the inability to work with tables/views without a single-column primary key. But there is, in my opinion, one strong benefit, lower code complexity and as a result, better maintainability of a software product. Hopefully, this simplicity will lead to a decreased failure rate of software development projects (at least those developed by the readers of php|architect!).

Further development of the ORM inside FaZend Framework will include stronger query optimization and integration with **Zend_Cache**. The project is open-source, and we encourage developers to join our team. For more information, e-mail us at team@fazend.com.

References

[Cerpa09] Narciso Cerpa et al., Why Did Your Project Fail? Communications of the ACM, vol.52, no.12, Dec 2009.

[Fowler08] M. Fowler et al., Patterns of Enterprise Application Architecture, Pearson 2008.

[Godoy08] A. V. Godoy, The Propel ORM, php|architect, vol. 7, issue 70, Oct 2008.

[Jorgensen06] M. Jorgensen et al., How large are software cost overruns? A review of the 1994 CHAOS report, Information and Software Technology, vol. 48, 2006.

[McNulty09] C. McNulty, A Tour of the Doctrine ORM, php|architect, vol. 8, issue 8, Aug 2009.

[OPhinney09] M. W. O'Phinney, Zend Framework, php|architect, vol. 8, issue 6, June 2009.

[POSA08] F. Buschmann et al., Pattern Oriented Software Architecture, A System of Patterns, Volume 1, Wiley 2008.

YEGOR BUGAYENKO is the lead architect of FaZend Framework and a proud holder of ZCE, ZFCE and PMP certificates. He is also the director and co-founder of TechnoPark Corp., a custom software development company specializing in complex and distributed web applications.