



Zend Framework Database Access

Bill Karwin



karwin software solutions



Introduction

- **What's in the Zend_Db component?**
- **Examples of using each class**
- **Using Zend_Db in MVC applications**

Introduction to Zend Framework

- **PHP class library**
- **Open source**
- **Emphasis on quality**
- **“Extreme simplicity”**
- **Popular**
- **Provides common web application functionality – so you don’t have to**
- **Use-at-will library of object-oriented PHP classes**
- **154,000+ lines of code**
- **100% PHP – works on standard PHP as well as Zend Core**
- **Requires PHP 5.1.4+**

Introduction to Zend Framework

- PHP class library
 - Open source
 - Emphasis on quality
 - “Extreme simplicity”
 - Popular
- Free software for development and deployment
 - New BSD license (similar to PHP itself)
 - Friendly to commercial projects
 - Code is free of legal claims, by virtue of the Contributor License Agreement

Introduction to Zend Framework

- PHP class library
 - Open source
 - **Emphasis on quality**
 - “Extreme simplicity”
 - Popular
- Agile development model
 - Frequent evaluation by large user community
 - Goal of 90% coverage in automated tests
 - Tests consist of 105,000+ lines of code (40% of total code)
 - Thorough documentation

Introduction to Zend Framework

- PHP class library
 - Open source
 - Emphasis on quality
 - “Extreme simplicity”
 - Popular
- Designed to make development easy
 - Solves 80% most needed use cases
 - Extensible; developers implement custom behavior from the remaining 20% (think of the “long tail”)

Introduction to Zend Framework

- PHP class library
 - Open source
 - Emphasis on quality
 - “Extreme simplicity”
 - Popular
- Over 2.3 M downloads – accelerating!
 - Over 300 contributors
 - Powers many high-profile web sites
 - IBM QEDWiki
 - Right Media Advertising Exchange
 - Varien Magento
 - *bwin* International
 - In-Ticketing
 - fav.or.it

What's in the Zend_Db component?

- **Database Adapters**
- **CRUD functions**
- **Quoting SQL values and identifiers**
- **Query profiler**
- **Query builder**
- **Table and Row OO patterns**
- **Common interface to multiple databases**
- **Supports both PDO and native database extensions:**
 - Mysqli
 - Oracle OCI8
 - IBM DB2
 - PDO IBM
 - PDO Mysql
 - PDO Mssql
 - PDO OCI
 - PDO Pgsq
 - PDO Sqlite

What's in the Zend_Db component?

- Database Adapters
- **CRUD functions**
- Quoting SQL values and identifiers
- Query profiler
- Query builder
- Table and Row OO patterns

Basic database operations

- insert()
- update()
- delete()
- query()
- fetchAll()
- etc.

What's in the Zend_Db component?

- Database Adapters
 - CRUD functions
 - Quoting SQL values and identifiers
 - Query profiler
 - Query builder
 - Table and Row OO patterns
- Encourages safer interpolation of values into SQL strings
 - Both values and SQL identifiers are supported
 - Not the same thing as SQL parameters – but those are supported too
 - Both quoting and SQL parameters help you to defend against SQL injection issues

What's in the Zend_Db component?

- Database Adapters
 - CRUD functions
 - Quoting SQL values and identifiers
 - Query profiler
 - Query builder
 - Table and Row OO patterns
- Measures duration of SQL queries
 - Filters by SQL statement type or by minimum duration
 - Reports recorded SQL statements and bound parameter values

What's in the Zend_Db component?

- Database Adapters
 - CRUD functions
 - Quoting SQL values and identifiers
 - Query profiler
 - Query builder
 - Table and Row OO patterns
- Procedural interface to create SELECT queries
 - Convenient when you need to “build” SQL with application logic
 - Helps you to produce valid SQL SELECT syntax

What's in the Zend_Db component?

- Database Adapters
 - CRUD functions
 - Quoting SQL values and identifiers
 - Query profiler
 - Query builder
 - Table and Row OO patterns
- Similar to *ActiveRecord* pattern in other frameworks
 - Flexible and extensible base classes for Tables, Rowsets, and Rows
 - Emphasis on supporting existing database schema; no restrictive conventions
 - Often used in Model classes in MVC apps

Database Adapters

Connecting to a database

- **Use the static factory() method of Zend_Db:**

```
require_once 'Zend/Db.php';  
$db = Zend_Db::factory('Mysqli',  
    array(  
        'host'      => 'localhost',  
        'dbname'   => 'test',  
        'username' => 'webappuser',  
        'password' => 'xxxxxxxx')  
);
```

- **Returns an object of Zend_Db_Adapter_Mysqli, which extends Zend_Db_Adapter_Abstract**

Using a configuration file

- **Write a configuration file, e.g. myconfig.ini:**

```
[staging]
```

```
database.adapter           = mysqli
```

```
database.params.host      = localhost
```

```
database.params.dbname    = test
```

```
database.params.username  = webappuser
```

```
database.params.password  = xxxxxxxx
```

- **Use Zend_Config to read config file:**

```
$config = new Zend_Config_Ini('myconfig.ini', 'staging');
```

- **Pass config object instead of plain values**

```
$db = Zend_Db::factory($config->database);
```

Running an SQL query & fetching results

- **The fetchAll() method returns an array of rows.**

```
$data = $db->fetchAll('SELECT * FROM bugs');
```

- **Each row is an associative array:**

```
foreach ($data as $row) {  
    echo $row['bug_description'];  
}
```

- **You can fetch rows one-by-one in a loop**
- **You can fetch rows as other structures**

Preparing an SQL query

- **Use the `prepare()` or `query()` methods to create a `Zend_Db_Statement` object:**

```
$stmt = $db->prepare(  
    'SELECT * FROM bugs  
    WHERE bug_status = ?')
```

- **Execute once, giving a parameter value:**

```
$stmt->execute(array('OPEN'));  
$openBugs = $stmt->fetchAll();
```

- **Execute a second time, giving another value:**

```
$stmt->execute(array('CLOSED'));  
$closedBugs = $stmt->fetchAll();
```

Inserting data

- **Use the insert() method.**
- **Pass the table name, and an associative array mapping columns to values:**

```
$db->insert( 'bugs',  
    array(  
        'bug_description' => 'help me',  
        'bug_status'      => 'NEW')  
    );
```

- **Get an auto primary key value (if applicable):**

```
$bugId = $db->lastInsertId();
```

Updating data

- Use the `update()` method.
- Pass the table name, an associative array mapping columns to new values, and an expression for the **WHERE** clause:

```
$n = $db->update( 'bugs',  
    array('bug_status' => 'NEW'),  
    'bug_id = 123'  
);
```

- Returns the number of rows affected.

Deleting data

- **Use the delete() method.**
- **Pass the table name, and an expression for the WHERE clause:**

```
$n = $db->delete('bugs', 'bug_id = 321');
```
- **Returns the number of rows affected.**

Retrieving table metadata

- **Use the describeTable() method:**

```
$bugsMetadata = $db->describeTable('bugs');
```

- **Returns an array indexed by column name:**

```
array(  
    'bug_id'           => array(...),  
    'bug_description' => array(...),  
    'bug_status'      => array(...),  
    'created_on'      => array(...),  
    'updated_on'      => array(...),  
    'reported_by'     => array(...),  
    'assigned_to'     => array(...),  
    'verified_by'     => array(...),  
)
```

Retrieving table metadata

- **Each value is an associative array of metadata for the respective column:**

```
'bug_id' => array(  
    'TABLE_NAME'           => 'bugs'  
    'COLUMN_NAME'         => 'bug_id'  
    'COLUMN_POSITION'     => 1  
    'DATA_TYPE'           => 'INTEGER'  
    'DEFAULT'             => null  
    'NULLABLE'            => false  
    'LENGTH'              => null  
    'SCALE'               => null  
    'PRECISION'           => null  
    'UNSIGNED'            => null  
    'PRIMARY'             => true  
    'PRIMARY_POSITION'    => 1  
    'IDENTITY'            => true  
),  
...entries for other columns follow...
```

Quoting SQL

Quoting SQL

- **Important when interpolating strings and PHP variables into SQL**
- **Not the same thing as query parameters**
 - Interpolated values add to a SQL query before prepare-time
 - Query parameters supply values to a prepared query at execute-time

Quoting SQL values

- **Use the `quote()` method to turn a string or variable into a quoted SQL string:**

```
$expr = $db->quote("O'Reilly");  
$stmt = $db->query("SELECT * FROM bugs  
WHERE bug_reporter = $expr");
```

Note: unlike `mysql_real_escape_string()`, the `quote()` method returns a string with quotes.

- `mysql_real_escape_string("O'Reilly")` returns: **O\<'Reilly**
- `$db->quote("O'Reilly")` returns: **'O\<'Reilly'**

Quoting SQL values in expressions

- **Use the `quoteInto()` method to substitute a scalar into a SQL expression:**

```
$whereExpr = $db->quoteInto('bug_reporter = ?',  
    "O'Reilly");  
$stmt = $db->query(  
    "SELECT * FROM bugs WHERE $whereExpr");
```

- **Results in the following query:**

```
SELECT * FROM bugs WHERE bug_reporter = 'O\'Reilly'
```

Quoting SQL identifiers

- **Identifiers are table names, column names, etc.**

- **Use the `quoteIdentifier()` method :**

```
$table = $db->quoteIdentifier('My Table');
```

```
$stmt = $db->query("SELECT * FROM $table WHERE ...");
```

- **Results in the following query:**

```
SELECT * FROM "My Table" WHERE ...
```

- **Helps if your identifiers contain whitespace, special characters, SQL keywords, or is case-sensitive.**

Query Profiler

Profiling SQL queries

- **Measures time to execute SQL queries**
- **Useful during development/debugging**
- **The Adapter has a Zend_Db_Profiler object, which is disabled by default**

```
$db->getProfiler()->setEnabled(true);
```
- **While the profiler is enabled, SQL queries are recorded, with the time it takes them to run.**

Profiling SQL queries: find longest query

```
$prof = $db->getProfiler();
$prof->setEnabled(true);

// run one or more queries with profiler enabled
$db->query(...); $db->insert(...); $db->update(...);

$max = 0;
foreach ($prof->getQueryProfiles() as $q) {
    if ($q->getElapsedSecs() > $max) {
        $max = $q->getElapsedSecs();
        $longestQuery = $q->getQuery();
    }
}
echo "The longest query ran $max seconds\n";
echo "SQL = $longestQuery\n";
```

Profiling SQL queries: filtering

- **Filtering queries to be recorded**

```
// By minimum duration:
```

```
$prof->setFilterElapsedSecs(5);
```

```
// By query type:
```

```
$prof->setFilterQueryType(Zend_Db_Profiler::SELECT  
| Zend_Db_Profiler::UPDATE);
```

- **Filtering queries to be reported**

```
$prof->getQueryProfiles(Zend_Db_Profiler::SELECT);
```

Query Builder

Building a SELECT query

- **Use the `select()` method of the Adapter to get an object of `Zend_Db_Select`:**

```
$select = $db->select();  
$select->from('bugs');  
$select->where('bug_status = ?', 'NEW');
```

- **Execute the query using the `Zend_Db_Select` object instead of a SQL string:**

```
$data = $db->fetchAll($select);
```

Building a SELECT query

- **The fluent usage can be convenient:**

```
$select = $db->select()->from('bugs')  
->where('bug_status = ?', 'NEW');
```

- **You can mix fluent and traditional usage.**
- **You can add SQL clauses in any order.**

```
$select = $db->select()->from('bugs')->order('bug_id');  
if ($condition == true) {  
    $select->joinNatural('other_table');  
}
```

Building a SELECT query: other clauses

- **Additional functions:**

- DISTINCT modifier
- Specify columns
- Table correlation names
- Specify SQL expressions
- Column aliases
- Joins to other tables
- GROUP BY clause
- HAVING clause
- ORDER BY clause
- LIMIT clause

```
$select = $db->select()  
->distinct()  
->from(array('a'=>'accounts'),  
       array('account_name',  
             'num_bugs'=>'COUNT(*)'))  
->joinLeft(array('b'=>'bugs'),  
          'b.reported_by =  
          a.account_name')  
->group('a.account_name')  
->having('num_bugs < 5')  
->order('a.account_name ASC')  
->limit(10, 20);
```



Table and Row Object-Oriented Patterns

Table Data Gateway

- Like the *ActiveRecord* pattern, this provides a simple OO interface for a database table.
- *Table Data Gateway* and *Row Data Gateway* patterns are based on Martin Fowler's "Patterns of Enterprise Architecture."
- Base classes for Table, Row, and Rowset are extensible, to allow you to define business logic.
- You may use Table objects in your MVC Model classes, or as your MVC Models.

Table Data Gateway: define a table

- **Start by defining a class for each table:**

```
class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name = 'bugs';
}
```

- **Default code is inherited from the base `Zend_Db_Table_Abstract` class.**
- **You don't need to define `$_name` if the class name matches the name of the table in the database.**

Table Data Gateway: instantiate a table

- **Option #1: pass the database Adapter object to the Table constructor:**

```
$bugsTable = new Bugs( $db );
```

- **Option #2: set a default Adapter for all Tables:**

```
Zend_Db_Table_Abstract::setDefaultAdapter($db);  
$bugsTable = new Bugs();
```

- **Option #3: save the Adapter object in Zend_Registry and reference it later by key:**

```
Zend_Registry::set('myDb', $db);  
$bugsTable = new Bugs( 'myDb' );
```

Table Data Gateway: query a table

- **Use the `fetchAll()` method to get a Rowset with all rows matching a condition you specify:**

```
$rowset = $bugTable->fetchAll("bug_status = 'OPEN'");
```

- **A Rowset object is iterable and countable.**
- **A Rowset is a collection of Row objects.**
- **A Row has an accessor for each column.**

```
foreach ($rowset as $row) {  
    echo "$row->bug_id: $row->bug_description\n";  
}
```

Table Data Gateway: find by key

- **Use the `find()` method with a primary key value or an array of values:**

```
$bugTable = new Bugs();  
$rowset1 = $bugTable->find(123);  
$rowset2 = $bugTable->find( array(123, 321) );
```

- **`$rowset1` contains 0 or 1 Row, `$rowset2` contains up to 2 Rows**
- **Tip: use the `current()` method to get the first row in a rowset:**

```
$row = $rowset1->current();
```

Table Data Gateway: UPDATE

- **Get the Row you want to update:**

```
$bugsTable = new Bugs();  
$rowset = $bugsTable->find(123);  
$row = $rowset->current();
```

- **Set a column value using the accessor:**

```
$row->bug_description = 'New description';
```

- **Use the `save()` method of the Row object to post the change to the database:**

```
$row->save();
```

Table Data Gateway: INSERT

- **Use the `createRow()` method of the Table object to get a blank Row:**

```
$bugsTable = new Bugs();  
$newRow = $bugsTable->createRow();
```

- **Set Row fields:**

```
$newRow->bug_description = 'help me';  
$newRow->bug_status = 'NEW';
```

- **Use the `save()` method of new Row object to post it to the database:**

```
$newRow->save();
```

Table Data Gateway: DELETE

- **Get a Row you want to delete:**

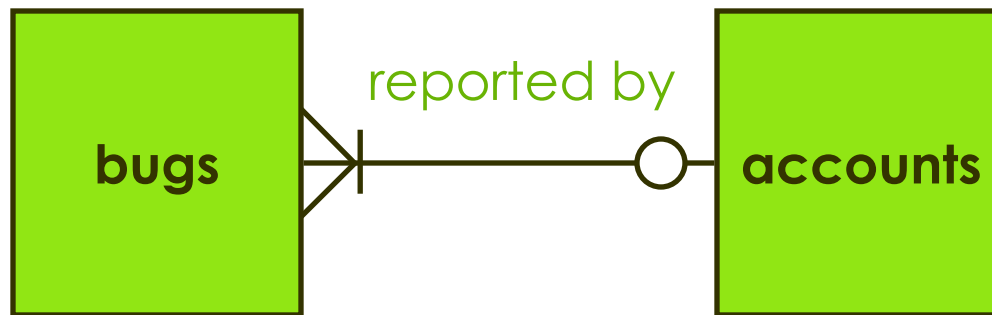
```
$bugsTable = new Bugs();  
$row = $bugsTable->find(123)->current();
```

- **Use the delete() method of the Row object:**

```
$row->delete();
```

Table Data Gateway: relationships

- Assume an Entity-Relationship like this:



- **Get a Row from the Accounts table:**

```
$accountsTable = new Accounts();  
$account = $accountsTable->find('bill')->current();
```

- **Find a Rowset in related table Bugs:**

```
$reportedBugs = $account->findBugs();
```

Table Data Gateway: relationships

- **You also can get the parent row.
Start with the dependent row in Bugs:**

```
$bugsTable = new Bugs();  
$bug = $bugsTable->find(123)->current();
```

- **Find the parent Row in related table Accounts:**

```
$reporter = $bug->findParentAccounts();
```

Table Data Gateway: relationships

- **Declare table relationships in the table class:**

```
class Accounts extends Zend_Db_Table_Abstract
{
    protected $_name = 'accounts';
}
class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name = 'bugs';
    protected $_referenceMap = array(
        'Reporter' => array(
            'columns'      => array('reported_by'),
            'refTableClass' => 'Accounts'
        )
    );
}
```

Table Data Gateway: customization

- **You can add custom logic to a Table class:**

```
class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name = 'bugs';

    public function insert(array $data)
    {
        if (empty($data['created_on'])) {
            $data['created_on'] = time();
        }
        return parent::insert($data);
    }
}
```

Using Zend_Db in MVC Applications

Using Zend_Db in MVC Applications

- **Each Model class in the MVC architecture encapsulates data and operations on data for a logical domain within your application.**
- **You can use either Zend_Db_Table objects or direct SQL queries to implement persistence for Model data.**
- **Common to define Models using an “is-a” relationship to Zend_Db_Table, but it can be more flexible to use a “has-a” relationship instead.**

Using Zend_Db in MVC Applications

- **Simple Models can extend directly from Zend_Db_Table_Abstract:**

```
class Countries extends Zend_Db_Table_Abstract
{
    protected $_name = 'countries';
}
```

- **Good if this Model corresponds one-to-one with a single physical database table.**

Using Zend_Db in MVC Applications

- **More complex Models use multiple Tables, or else Db Adapter methods to execute SQL.**

```
class QuarterlyReport // extends no base class
{
    protected $_salesTable;
    protected $_productsTable;
    protected $_dbAdapter; // for direct SQL queries
    public function getReport($quarter) { ... }
}
```

- **Good if you want this Model's interface to be decoupled from the physical database structure.**

Recap of Zend_Db

- **Adapters to PHP database extensions**
- **Quoting SQL identifiers and values**
- **Query profiler**
- **SELECT query builder**
- **Table Data Gateway & Row Data Gateway**



Thanks

<http://framework.zend.com/>



Copyright © 2007, Zend Technologies Inc.