Database()

Contents:

Overview

The Database() class intends to be a complete, powerful, simple and fast MySQL database wrapper, written in pure PHP, with good scaling options and flexible configuration options.

You can use it to write a database-driven website with a few lines of code, and scale it easily just adding servers to your Database() instance.

Methods

You'll have to instance Database() to access any method listed here. Singleton is not supported.

Standard methods:

- **`$instance->connect(void)`** connects to the MySQL backend using configuration values (see “Code examples” below). Returns a boolean value (enable debug to see error messages if **`false`** is returned)

- **`$instance->close(void)`** closes any open connection to the MySQL backend

- **`$instance->escape(string)`** returns an escaped string (using the charset preferences of a write or read/write server)

- **`$instance->query($sql)`** runs the query (if balancing is enabled, the query is sent to the corresponding server) and returns a MySQL Resource

Fetching methods:

- **`$instance->fetch(query or fields[, table, where, limit])`** builds an SQL query if needed and returns an array of objects, or false in case of error (errors sent to debug)

- **`$instance->fetch_row(query or fields[, table, where, limit])`** does the same than `$instance->fetch`, but returns an object (just one row)

- **`$instance->fetch_one(query or field[, table, where, limit])`** does (also) the same, but returns a string (just a field, for example, a `COUNT`)

ActiveRecord Insertions:

- **`$instance->add(table)`** returns an object for adding values to the database. See “Code examples” below for more information.

- **`$instance->modify(table)`** returns an object for modifying values in a database. See “Code examples” below for more information.

Other methods:

- **`$instance->insert(table, values)`** adds the array of key/values to the table specified

- **`$instance->update(table, values[, where])`** updates the specified table with the array of key/values, also matching the where condition (it can be a key/value array or just an integer, in which case the field name will be assumed `id`)

Load balancing features

The most common way to do load balancing with MySQL is to set up a master server and a set of slave servers. Then, every query is sent to a random slave server if it’s a read query (SELECT), or to a master server if it’s a write (INSERT, UPDATE) query.

Usually, database-driven websites have to be partially rewritten to split queries between servers when they grow too much. Database() can do this work for you: just enable balancing mode with a line of code and add more servers to your configuration.

If you enable balancing mode in Database(), read queries will be sent to a random slave server and write queries will be sent to a random master server. You can also configure it to send each write queries to all servers (for example, each UPDATE or INSERT is sent to all write servers).

Master-master replication is also supported: if balancing is disabled (balancing mode just splits queries between master and slave servers) and an array of servers is supplied, Database() will pick one randomly.

Multiple servers (master-master scheme), no read/write separation:

```
$db = new Database;

$db->hostname = array(‘mysql1.mycompany’, ‘mysql2.mycompany’);
$db->username = ‘root’;
```

```
$db->password = ‘password’;
$db->database = ‘mydbname’;

$db->connect() or exit(‘Something went wrong’);
```

Note that when using balancing, the MySQL username and password must be the same for all servers.

In this example, the connect() method will randomly pick one server and connect to. If the server is down, it will connect to the next available server (also randomly picked).

Master/slave replication:

```
$db = new Database;

$db->hostname = array(array(‘master’, ‘master.mycompany’),
array(‘slave’, ‘slave.mycompany’));
$db->username = ‘root’;
$db->password = ‘password’;
$db->database = ‘mydbname’;

$db->balance = true;

$db->connect() or exit(‘Something went wrong’);
```

You’ll notice that when balancing is enabled (`$db->balance = true`), `$db->hostname` value is an array of arrays, containing the server type as first value.

At least one master server and a slave server should be given (in this format), or a fatal error will be thrown (and that will abort the entire page if `$db->strict` is on).

Every query ran by Database() is sent to it’s corresponding server. For example, if you issue an update or an insertion directly (`$db->query(“UPDATE table SET x=y”)`), the query is sent to a random master server, or to all master servers if `$db->balance_iter` is enabled. Let’s see what’s that.

Balancing with multiple masters and multiple slaves:

As your website grows, you’ll have to add more master servers. Database() always picks one master server and one slave server (randomly) each time you connect. But what if you want to send write queries (updates, insertions) to all servers?

```
$db = new Database;

$db->hostname = array(array(‘master’, ‘master.mycompany’),
array(‘master’, ‘master2.mycompany’), array(‘slave’,
‘slave.mycompany’), array(‘slave’, ‘slave2.mycompany’));
$db->username = ‘root’;
$db->password = ‘password’;
$db->database = ‘mydbname’;

$db->balance = true;
$db->balance_iter = true;
```

```
$db->connect() or exit('Something went wrong');

$users = $db->fetch("username, realname", "users");

$insertion = $db->add('users');
$insertion->username = 'anusername';
$insertion->realname = 'John Doe';
$insertion->save();
```

The first block of code connects, using an array of servers, which contains two master servers and two slave servers. Then, balancing is enabled and `balance_iter` is turned on.

The `users` database is read. The query is automatically built, and sent to the slave server picked by `connect()`.

Then, a new row is inserted (using `$db->add`). Because `$db->balance_iter` is enabled, the query will be sent to **all** master servers.

Code examples

Initialize:

```
require('Database.php');

$db = new Database;
$db->hostname = 'localhost';
$db->username = 'root';
$db->password = 'password';
$db->database = 'mydbname';
$db->connect() or exit('Something went wrong');
```

That's the simplest way to use Database().

Fetching data:

Those code blocks do the same:

```
print_r($db->fetch("SELECT username, realname FROM users"));
print_r($db->fetch("username, realname", "users"));

print_r($db->fetch_row("SELECT * FROM users LIMIT 1"));
print_r($db->fetch_row("*", "users", false, 1));

print $db->fetch_one("SELECT realname FROM users LIMIT 1");
print $db->fetch_one("realname", "users", false, 1);
```

Insertions and modifications using ActiveRecord:

Database() implements a partial ActiveRecord functionality for insertions and updates.

Insertions:

```
$i = $db->modify(‘users’); # creates an object for the given table
$i->myrow = ‘myvalue’;

$i->save(); # no parameters needed!
```

Modifications:

```
$m = $db->modify(‘users’);
$m->myrow = ‘mynewvalue’;

$modification->save(2); # `id’ row value (integers only)
```

or

```
$modification->save(array(‘fieldname’ => ‘matchingvalue’));
```

If you don’t specify a WHERE delimitation, the UPDATE will affect the whole table.

MySQL variables:

MySQL provides a set of variables, for example `affected_rows`, `insert_id` or `num_rows`. It’s really easy to use those variables with Database()!

```
$users = $db->fetch(“username, realname”, “users”);
print $db->num_rows;

$insertion = $db->add(‘users’);
$insertion->username = ‘anusername’;
$insertion->realname = ‘John Doe’;
$insertion->save();
print $db->insert_id;

$modification = $db->modify(‘users’);
$modification->realname = ‘Doe Jr.’;
$modification->save($db->insert_id);
print $db->affected_rows;
```

In this example, we retrieve the entire users table as an array of objects, print the ID of that insertion, modify that insertion (using `insert_id`), changing the `realname` field to ‘Doe Jr.’ and print the number of rows affected by that modification (UPDATE).

Built SQL queries and WHERE statments:

In the example above, we do a modification using the super-simple ActiveRecord implementation. But what if the name of our primary field isn’t **`id`**, or we want to use a different WHERE condition?

Just use an associative array:

```
$modification = $db->modify(‘users’);
$modification->realname = ‘Doe Jr.’;
```

```
    $modification->save(array('username'=>'anusername'));
    print $db->affected_rows;
```

This also applies for `$db->fetch`, `$db->fetch_row` or `$db->fetch_one` statments.

Avoiding built SQL queries:

Sometimes you want to use complex SQL queries, or just build queries on your own side. You can do that, and it's very easy.

For example:

```
    $users = $db->fetch("SELECT username, realname FROM users");
```

Will work the same way that:

```
    $users = $db->fetch("username, realname", "users");
```

For other insertions, just use `$db->query`:

```
    $db->query("UPDATE users SET realname='Doe Jr.' WHERE
username='anusername'");
```

Configuration values

- `(bool)` **`$db->debug`** enable or disable in-screen debug messages
- `(bool)` **`$db->strict`** if enabled, aborts the execution of the entire program when a fatal error is raised
- `(bool)` **`$db->balance`** enable or disable balancing (see "Load balancing features" section)
- `(bool)` **`$db->balance_iter`** if enabled, sends every write query (INSERT, UPDATE) to each master server given in `$db->hostname`

Security

Security is a big point on most web-based applications, and Database() tries to keep things secure in most cases.

All ActiveRecord insertion or modification keys and values are safely escaped, much like WHERE statments in built queries (using `$db->fetch`, `$db->fetch_row`, etc.)

On the other side, table names, field names or limits should be properly escaped, as those values don't pass any control and are directly sent to the MySQL server (for example, passing a bad formatted LIMIT would generate a malformed SQL query).

So, in a few words: WHERE values are secure, and you shouldn't let the user interact with any LIMIT, field selection or table name (or if you have to, keep an eye in security!).

More information

Database() intends to be an efficient database wrapper with a large set of features, with a main target of improving development time on little and big projects, in part avoiding partial code rewrites on big-growing websites or SQL chunks in little websites.

No new features are planned, only security and compatibility updates will take place (infrequent updates). Database() is built as a common, full-featured database wrapper (to build things in its top), not as an evolving tool.

Code is released under the GNU Lesser General Public License (A.K.A. LGPL), version 3.