Database

Overview

HUBzero has been built with the ability to use several different kinds of SQL-database-systems and to run in a variety of environments with different table-prefixes. In addition to these functions, the class automatically creates the database connection. Besides instantiating the object, at a minimum, you only need 2 lines of code to get a result from the database in a variety of formats. Using the database layer ensures a maximum of compatibility and flexibility for your extension.

This tutorial looks at how to set and execute various queries.

Configuration

The database configuration for a hub is located at app/config/database.php.

Connections

A hub establishes a database connection, by default, with the configuration specified in app/config/database.php but alternate connections may be established.

```
$mydb = HubzeroDatabaseDriver::getInstance([
    'driver'   => 'pdo',
    'host'     => 'example',
    'user'     => 'example',
    'password' => '******',
    'database' => 'mystuff',
    'prefix'   => 'hub_'
]);
```

Preparing The Query

```
// Get a database object
$db = App::get('db');

$query = "SELECT * FROM `#__example_table` WHERE `id` = 999999;";
$db->setQuery($query);
```

First we instantiate the database object, then we prepare the query. You can use the normal SQL-syntax, the only thing you have to change is the table-prefix. To make this as flexible as possible, Joomla! uses a placeholder for the prefix, the "#__". In the next step, the $db->setQuery(), this string is replaced with the correct prefix.

Now, if we don't want to get information from the database, but insert a row into it, we need one more function. Every string-value in the SQL-syntax should be quoted. For example, MySQL uses back-ticks `` for names and single quotes '' for values. Joomla! has some functions to do this for us and to ensure code compatibility between different databases. We can pass the names to the function $db->quoteName($name) and the values to the function $db->Quote($value).

A fully quoted query example is:

```
$query = "
  SELECT *
    FROM ".$db->nameQuote('#__example_table')."
    WHERE ".$db->nameQuote('id')." = ".$db->quote('999999').";
  ";
```

Whatever we want to do, we have to set the query with the $db->setQuery() function. Although you could write the query directly as a parameter for $db->setQuery(), it's commonly done by first saving it in a variable, normally $query, and then handing this variable over. This helps writing clean, readable code.

setQuery($query)

The setQuery($query) method sets up a database query for later execution either by the query() method or one of the Load result methods.

```
$db = App::get('db');
$query = "/* some valid sql string */";
$db->setQuery($query);
```

Note: The parameter $query must be a valid SQL string, it can either be added as a string parameter or as a variable; generally a variable is preferred as it leads to more legible code and can help in debugging.

setQuery() also takes three other parameters: $offset, $limit - both used in list pagination; and $prefix - an alternative table prefix. All three of these variables have default values set and can

usually be ignored.

Executing The Query

To execute the query, Joomla! provides several functions, which differ in their return value.

Basic Query Execution

The query() method is the the basic tool for executing sql queries on a database. In the CMS it is most often used for updating or administering the database and not seen often for loading data. This largely because the various load methods detailed on this page have the query step built in to them.

The syntax is very straightforward:

```
$db = App::get('db');
$query = "/* some valid sql string */";
$db->setQuery($query);
$result = $db->query();
```

Note: $db->query() returns an appropriate database resource if successful, or FALSE if not.

Query Execution Information

- getAffectedRows()
- explain()
- insertid()

Insert Query Execution

- insertObject()

Query Results

The database class contains many methods for working with a query's result set.

Single Value Result

loadResult()

Use loadResult() when you expect just a single value back from your database query.

id	name	email	username
1	John Smith	johnsmith@example.com	johnsmith
2	Magda Hellman	magda_h@example.com	magdah
3	Yvonne de Gaulle	ydg@example.com	ydegaulle

This is often the result of a 'count' query to get a number of records:

```
$db = App::get('db');
$query = "
  SELECT COUNT(*)
    FROM ".$db->nameQuote('#__my_table')."
    WHERE ".$db->nameQuote('name')." = ".$db->quote($value).";
  ";
$db->setQuery($query);
$count = $db->loadResult();
```

or where you are just looking for a single field from a single row of the table (or possibly a single field from the first row returned).

```
$db = App::get('db');
$query = "
  SELECT ".$db->nameQuote('field_name')."
    FROM ".$db->nameQuote('#__my_table')."
    WHERE ".$db->nameQuote('some_name')." = ".$db->quote($some_va
lue).";
  ";
$db->setQuery($query);
$result = $db->loadResult();
```

Single Row Results

Each of these results functions will return a single record from the database even though there may be several records that meet the criteria that you have set. To get more records you need

to call the function again.

id	name	email	username
1	John Smith	johnsmith@example.com	johnsmith
2	Magda Hellman	magda_h@example.com	magdah
3	Yvonne de Gaulle	ydg@example.com	ydegaulle

loadRow()

loadRow() returns an indexed array from a single record in the table:

```
...
$db->setQuery($query);
$row = $db->loadRow();
print_r($row);
```

will give:

```
Array (
 [0] => 1
 [1] => John Smith
 [2] => johnsmith@example.com
 [3] => johnsmith
)
```

You can access the individual values by using:

```
$row['index'] // e.g. $row['2']
```

Note:

1. The array indices are numeric starting from zero.
2. Whilst you can repeat the call to get further rows, one of the functions that

returns multiple rows might be more useful

loadAssoc()

loadAssoc() returns an associated array from a single record in the table:

```
$db->setQuery($query);
$row = $db->loadAssoc();
print_r($row);
```

will give:

```
Array (
 [id] => 1
 [name] => John Smith
 [email] => johnsmith@example.com
 [username] => johnsmith
)
```

You can access the individual values by using:

```
$row['name'] // e.g. $row['name']
```

Whilst you can repeat the call to get further rows, one of the functions that returns multiple rows might be more useful

loadObject()

loadObject() returns a PHP object from a single record in the table:

```
$db->setQuery($query);
$result = $db->loadObject();
print_r($result);
```

will give:

```
stdClass Object (
 [id] => 1
 [name] => John Smith
 [email] => johnsmith@example.com
 [username] => johnsmith
)
```

You can access the individual values by using:

```
$row->index // e.g. $row->email
```

Whilst you can repeat the call to get further rows, one of the functions that returns multiple rows might be more useful

Single Column Results

Each of these results functions will return a single column from the database.

id	name	email	username
1	John Smith	johnsmith@example.com	johnsmith
2	Magda Hellman	magda_h@example.com	magdah
3	Yvonne de Gaulle	ydg@example.com	ydegaulle

loadColumn()

loadColumn() returns an indexed array from a single column in the table:

```
$query = "
  SELECT name, email, username
    FROM . . . ";

$db->setQuery($query);
$column= $db->loadColumn();
print_r($column);
```

will give:

```
Array (
 [0] => John Smith
 [1] => Magda Hellman
 [2] => Yvonne de Gaulle
)
```

You can access the individual values by using:

```
$column['index'] // e.g. $column['2']
```

Note:

1. The array indices are numeric starting from zero.
2. loadColumn() is equivalent to loadcolumn(0)

loadColumn($index)

loadColumn($index) returns an indexed array from a single column in the table:

```
$query = "
  SELECT name, email, username
    FROM . . . ";

$db->setQuery($query);
$column= $db->loadColumn(1);
print_r($column);
```

will give:

```
Array (
 [0] => johnsmith@example.com
 [1] => magda_h@example.com
 [2] => ydg@example.com
)
```

You can access the individual values by using:

```
$column['index'] // e.g. $column['2']
```

loadColumn($index) allows you to iterate through a series of columns in the results

```
$db->setQuery($query);
for ( $i = 0; $i <= 2; $i++ ) {
  $column= $db->loadColumn($i);
  print_r($column);
}
```

will give:

```
Array ( [0] => John Smith [1] => Magda Hellman [2] => Yvonne de G
aulle )
Array ( [0] => johnsmith@example.com [1] => magda_h@example.com [
2] => ydg@example.com )
Array ( [0] => johnsmith [1] => magdah [2] => ydegaulle )
```

The array indices are numeric starting from zero.

Multi-Row Results

Each of these results functions will return multiple records from the database.

id	name	email	username

id	name	email	username
1	John Smith	johnsmith@example.com	johnsmith
2	Magda Hellman	magda_h@example.com	magdah
3	Yvonne de Gaulle	ydg@example.com	ydegaulle

loadRowList()

loadRowList() returns an indexed array of indexed arrays from the table records returned by the query:

```
$db->setQuery($query);
$row = $db->loadRowList();
print_r($row);
```

will give:

```
Array (
 [0] => Array ( [0] => 1 [1] => John Smith [2] => johnsmith@example.com [3] => johnsmith )
 [1] => Array ( [0] => 2 [1] => Magda Hellman [2] => magda_h@example.com [3] => magdah )
 [2] => Array ( [0] => 3 [1] => Yvonne de Gaulle [2] => ydg@example.com [3] => ydegaulle )
)
```

You can access the individual values by using:

```
$row['index'] // e.g. $row['2']
```

and you can access the individual values by using:

```
$row['index']['index'] // e.g. $row['2']['3']
```

The array indices are numeric starting from zero.

loadAssocList()

loadAssocList() returns an indexed array of associated arrays from the table records returned by the query:

```
$db->setQuery($query);
$row = $db->loadAssocList();
print_r($row);
```

will give:

```
Array (
 [0] => Array ( [id] => 1 [name] => John Smith [email] => johnsmi
th@example.com [username] => johnsmith )
 [1] => Array ( [id] => 2 [name] => Magda Hellman [email] => magd
a_h@example.com [username] => magdah )
 [2] => Array ( [id] => 3 [name] => Yvonne de Gaulle [email] => y
dg@example.com [username] => ydegaulle )
)
```

You can access the individual rows by using:

```
$row['index'] // e.g. $row['2']
```

and you can access the individual values by using:

```
$row['index']['column_name'] // e.g. $row['2']['email']
```

loadAssocList($key)

loadAssocList($key) returns an associated array - indexed on 'key' - of associated arrays from the table records returned by the query:

```
$db->setQuery($query);
$row = $db->loadAssocList('username');
print_r($row);
```

will give:

```
Array (
 [johnsmith] => Array ( [id] => 1 [name] => John Smith [email] =>
 johnsmith@example.com [username] => johnsmith )
 [magdah] => Array ( [id] => 2 [name] => Magda Hellman [email] =>
 magda_h@example.com [username] => magdah )
 [ydegaulle] => Array ( [id] => 3 [name] => Yvonne de Gaulle [ema
il] => ydg@example.com [username] => ydegaulle )
)
```

You can access the individual rows by using:

```
$row['key_value'] // e.g. $row['johnsmith']
```

and you can access the individual values by using:

```
$row['key_value']['column_name'] // e.g. $row['johnsmith']['email
']
```

Note: Key must be a valid column name from the table; it does not have to be an Index or a Primary Key. But if it does not have a unique value you may not be able to retrieve results reliably.

loadObjectList()

loadObjectList() returns an indexed array of PHP objects from the table records returned by the query:

```
$db->setQuery($query);
$result = $db->loadObjectList();
print_r($result);
```

will give:

```
Array (
 [0] => stdClass Object ( [id] => 1 [name] => John Smith
     [email] => johnsmith@example.com [username] => johnsmith )
 [1] => stdClass Object ( [id] => 2 [name] => Magda Hellman
     [email] => magda_h@example.com [username] => magdah )
 [2] => stdClass Object ( [id] => 3 [name] => Yvonne de Gaulle
     [email] => ydg@example.com [username] => ydegaulle )
)
```

You can access the individual rows by using:

```
$row['index'] // e.g. $row['2']
```

and you can access the individual values by using:

```
$row['index']->name // e.g. $row['2']->email
```

loadObjectList('key')

loadObjectList('key') returns an associated array - indexed on 'key' - of objects from the table records returned by the query:

```
$db->setQuery($query);
$row = $db->loadObjectList('username');
print_r($row);
```

will give:

```
Array (
 [johnsmith] => stdClass Object ( [id] => 1 [name] => John Smith
     [email] => johnsmith@example.com [username] => johnsmith )
 [magdah] => stdClass Object ( [id] => 2 [name] => Magda Hellman
     [email] => magda_h@example.com [username] => magdah )
 [ydegaulle] => stdClass Object ( [id] => 3 [name] => Yvonne de G
aulle
     [email] => ydg@example.com [username] => ydegaulle )
)
```

You can access the individual rows by using:

```
$row['key_value'] // e.g. $row['johnsmith']
```

and you can access the individual values by using:

```
$row['key_value']->column_name // e.g. $row['johnsmith']->email
```

Note: Key must be a valid column name from the table; it does not have to be an Index or a Primary Key. But if it does not have a unique value you may not be able to retrieve results reliably.

Misc Result Set Methods

getNumRows()

getNumRows() will return the number of result rows found by the last query and waiting to be read. To get a result from getNumRows() you have to run it after the query and before you have retrieved any results.

```
$db->setQuery($query);
$db->query();
$num_rows = $db->getNumRows();
```

```
print_r($num_rows);
$result = $db->loadRowList();
```

will give:

3

Note: if you run getNumRows() after loadRowList() - or any other retrieval method - you may get a PHP Warning.

Query Builder

Introduction

HUBzero offers a query builder to help in the abstraction of language-specific SQL syntax, as well as making code more readable. While the query builder itself takes many of its structural queues from SQL, it is not syntax specific. To get started, simply:

```
$query = new HubzeroDatabaseQuery;

// Sample select statement
$users = $query->select('*')
               ->from('#__users')
               ->whereEquals('job', 'programmer')
               ->fetch();
```

Functionally speaking, the query builder serves as a nice intermediary between the feature-rich abstraction that is the ORM, and the execution of raw language specific queries against the database driver. The query builder is actually available on ORM models, and the models will filter all applicable method calls down from the model itself to the underlying query as needed.

Fetching Results

Caching

By default, when fetching results using the query builder, query results will be cached. To disable this behavior, you can pass TRUE as the second argument of the fetch method call.

```
$query->fetch('rows', true);
```

Alternatively, instead of disabling the cache for a single fetch, you can clear the entire cache.

```
Query::purgeCache();
```

Inserting, Updating, and Deleting

When it comes to adding, modifying, or removing records, you have two options for going about this. You can manually build the appropriate query, or you can use one of the shortcut methods. To exemplify this behavior, check out the following two examples.

```
// Full insert
$query = new Query;

$query->insert('users')
      ->values(['name' => 'me', 'email' => 'you@me.com'])
      ->execute();

// Shortcut method
$query->push('users', ['name' => 'me', 'email' => 'you@me.com']);
```

The same principle also applies to updates and deletes:

```
// Full update
$query = new Query;

$query->update('users')
      ->set(['name' => 'you'])
      ->whereEquals('id', 1)
      ->execute();

// Shortcut method
$query->alter('users', 'id', 1, ['name' => 'you']);
```

```
// Full delete
$query = new Query;

$query->delete('users')
      ->whereEquals('id', 1)
      ->execute();

// Shortcut method
$query->remove('users', 'id', 1);
```

Migrations

Overview

HUBzero offers the muse command for automating and simplifying common web developer and system administrator tasks. Of those tasks, running database and content migrations is probably the most crucial to the successful management and deployment of new and updated HUBzero extensions.

The following sections assume that you have the muse command in your path and can execute the script directly. If that is not the case, replace all calls to muse with /www/yourdocroot/muse.

In addition to this documentation, more detailed documentation can always be found by calling: muse migration help

Running Migrations

Running migrations in its basic form is rather simple (though there are a plethora of options available to complicate things if you so desire). Simply type muse migration to run migrations in dry-run mode. This will tell you if you have any pending migrations to run, or if you have perhaps missed a previous migration. If satisfied with what migrations thinks needs to be done, simply run muse migration -f to run the full migration.

That's it!

OK, there's more. By default, migrations won't run migrations that have been missed in the past. To tell migrations to run all pending migrations, irrelevant of date, include the -i option. All other available options can be found by running muse migration help as mentioned above.

Creating Your Own Migrations

This is where the fun begins...

Creating migrations is essential to anyone deploying new extensions in an environment where database tinkering on prod is frowned upon. If the idea of production database access doesn't send chills down your spine, then at least having a migration written will offer a well documented change log for your extensions.

Muse has some basic commands for scaffolding, one of which allows you to create a template migration. To get this auto-generated goodness for yourself, type muse scaffolding create migration -e=extension_name. Here, the extension name would be the extension you are working on, in the form of com_mycomponent or plg_stuff_coolthing. This will drop you into your default editor with the template migration in place and setup according to the HUBzero

conventions of naming and layout.

```
<?php

use HubzeroContentMigrationBase;

// No direct access
defined('_HZEXEC_') or die();

/**
 * Migration script for content
 **/
class Migration20160924000001ComGroups extends Base
{
    /**
     * Up
     **/
    public function up()
    {
        // Changes to be made ...
    }

    /**
     * Down
     **/
    public function down()
    {
        // Reverse the changes ...
    }
}
```

The migration command will manage what migrations have been run and in what environment. That way you don't have to worry about what you've run and where. That being said, we think it's generally a good idea to make your migrations as foolproof and backwards compatible as possible. To that end, we've added a handful of helper functions to make things as simple as possible. These functions are available on the database object inside of your migration. They are as follows:

```
$this->db->ifTableExists('tableName');
$this->db->ifTableHasField('tableName', 'fieldName');
$this->db->ifTableHasKey('tableName', 'keyName');
```

As an example, instead of just blindly executing an alter table statement to add a new column, you might instead wrap the execution of that statement if an if block that checks for the existence of the table, and the non-existence of the field you want to add...like so:

```
class Migration20160924000001ComGroups extends Base
{
    /**
     * Up
     **/
    public function up()
    {
        if ($this->db->tableExists('myTable') && !$this->db->tableHasF
ield('myTable', 'myNewField'))
        {
            //...
        }
    }
}
```

Feel free to glance at other migrations in /www/your_doc_root/core/migrations for sample usage.

We've also started adding some additional features to make generating your migrations even easier. So, for example, if you're writing a migration to generate a new table, you can now do muse scaffolding create migration for jos_table_name -e=extension_name. This will create the migration as before, but this time, the migration is completely written for you! We'll add more info here as new features are developed. Also note, the extension name is still required at this time, as the table name and extension name are not explicitly related.

Working with Extensions

Within migrations, there are several helper methods available for common tasks. These methods are also valuable as they abstract out many of the idiosyncrasies of different versions of the database.

These methods primarily include interacting with the extensions tables:

```
$this->addComponentEntry($name);
$this->addPluginEntry($folder, $element);
$this->addModuleEntry($element);
$this->enablePlugin($folder, $element);
$this->disablePlugin($folder, $element);
```

The counter methods also exist for deleting components, plugins, and modules. These methods are structured the same, simply replace add* with delete*.

Showing Progress

When working with migrations and writing your own, you may occasionally find yourself needing to write a computationally intensive and potentially lengthy migration. When you do this, it's helpful to provide the end user of your migration (maybe other developers or customers) with information on the progress of the migration - rather than making them wonder if something has gone horribly wrong. To accomplish this, you can use the progress callback available on the migration class.

Initialize the progress tracker to show a message about what you're doing.

```
$this->callback('progress', 'init', array('Running ' . __CLASS__ . '.p
hp:'));
```

Next, occasionally update the progress notification with your current status.

```
// $i here is a number between 1 and 100 for percentage-based progress
$this->callback('progress', 'setProgress', array($i));
```

Finally, when all is said and done, clean up.

```
$this->callback('progress', 'done');
```

Lastly, instead of a percentage based progress tracker, you can also show a ratio based notification.

```
// Here we have an example of a ratio with 25 total items (the denomin
```

```
ator)
$this->callback('progress', 'init', array('Running ' . __CLASS__ . '.p
hp:', 'ratio', 25));

// Update to 4 (numerator) out of 25 complete
$this->callback('progress', 'setProgress', array(4, 25));

// All done
$this->callback('progress', 'done');
```

Logging Messages

When running migrations, it can be helpful to display information about what is being changed and indicating what did or didn't work in a migration. To accomplish this, you can use the log callback available on the migration class.

```
$this->callback('migration', 'log', array('message' => 'Adding new tab
le `foo`', 'type' => 'info'));
```

Available pre-defined log types are info, success, warning, error. Messages will be formatted/colored based on the log type.

For versions 2.2.15+, there's a log() method on the base migration class to simplify things.

```
// Log info
$this->log('Adding new table `foo`', 'info');

// Log success
$this->log('Migration of data to new table `foo` successfully complete
d', 'success');

// Log warning
$this->log('Dropping table `foo`', 'warning');

// Log error
$this->log('Failed to move data to new table `foo`', 'error');
```

The default message type is info so the second argument can be left off for purely information log messages.

```
$this->log('Adding new table `foo`');
```

Note: The log() method is only available in versions 2.2.15+

ORM

Introduction

Object Relational Mapping, or ORM, is a common paradigm found in many modern CMSs. HUBzero is no exception. HUBzero's ORM is similar to that of many other frameworks, and should be easy to pick up if you've had some experience elsewhere. The goal of HUBzero's ORM is to increase commonality and uniformity in code, while decreasing barriers to entry and errors for developers. While the ORM may not do everything you need it to in some extreme cases, it is relatively full-featured and should greatly speed up the development process.

The Basics

Accessing data

Getting started with the ORM is really quite simple. Your first model could be as basic as an empty class. Consider a User model as our primary working example:

```
use HubzeroDatabaseRelational;

class User extends Relational
{
}
```

With that model in place, you could then loop through all users:

```
foreach (User::all() as $user)
{
    echo $user->name;
}
```

The relational models implement the IteratorAggregate interface, which means that when you start to loop over a model, it will automatically fetch the results for you. If you don't want to loop over the results, you should explicitly tell it to fetch the rows.

```
$users = User::all()->rows();
```

To access a single user, you can load it up by it's primary key. To do so, use one of the "one*" methods.

```
$user = User::one($id);
// or
$user = User::oneOrFail($id);
// or
$user = User::oneOrNew($id);
```

Using the oneOrFail method will result in an exception being thrown if the user ID provided does not exist. The oneOrNew method will result in a blank user being returned if the provided ID does not exist.

In addition to the one*() or all() methods, you also have access to the query builder methods for programmatically limiting the results based on SQL-like constrains. For example:

```
$users = User::whereEquals('name', 'Me');
```

Creating, updating, and deleting

Saving and deleting with the ORM is easy!

```
// Saving/creating a user
$user = oneOrNew(1);
$user->set('name', 'New User');
$user->set('email', 'awesome@gmail.com');
$user->save();

// Deleting a user
$user = oneOrFail(1);
$user->destroy();
```

Don't worry, the relational models will figure out whether you are creating a new model or updating an existing one.

Relationships

Defining relationships between models is a key element to ORMs. The HUBzero ORM offers many standard relationships, including one-one, one-many, and many-many.

One to One

One to one relationships are the simplest variety. To create a one to one relationship between our previous User model and a Phone model, we would add the following method to our class:

```
public function phone()
{
    return $this->oneToOne('Phone');
}
```

Internally, this will attempt to join the Users table to the phone table by way of the user.id and phone.user_id keys. These keys can be overwritten by passing a second or third parameter to the oneToOne method.

It's also important to note that, if you're referencing a model in a different namespace than the current model, you should include the full name-spaced classname.

And to use that relationship, you could do something like this:

```
$user  = User::oneOrFail(1);
$phone = $user->phone;
```

The HUBzero ORM offers dynamic properties. This allows you to simply access the faux property phone, instead of having to explicitly call the phone() method defined above.

One to Many

One to many relationships, though slightly more complex, are probably the most popular relationship scenario. A user may author many posts, or upload many files, or make many comments, and so on. To define a one to many relationship, we:

```
public function posts()
{
    return $this->oneToMany('Post');
}
```

This will join the users table to the posts table by way of users.id and posts.user_id, similar to the way that the one to one relationship works. But, instead of returning a single model, this will return a rows collection of models.

To use this data, you might do something like:

```
$user  = User::oneOrFail(1);
foreach ($user->posts as $post)
{
    echo $post->content;
}
```

Belongs to One

The belongs to one relationship is the inverse of the one to one and one to many relationships. It functions in the same manner as the one to one relationship, but reverses the connection direction (and consequently also reverses the key parameters accepted by the belongsToOne method). Using the above example, a belongs to one relationship between posts and users might look as follows.

```
public function user()
{
    return $this->belongsToOne('User');
}
```

Many to Many

The many to many relationship introduces the relational idea of an an associative entity. Here, we are often trying to mimic a structure such as user roles. Consider the follow table structure.

roles	**users**	**role_user**
id	id	id
name	name	role_id
permissions	email	user_id

To accomplish this many to many structure, of users having multiple roles, and a role being assignable to multiple users, you would structure your models like this.

```
// in users model
public function roles()
{
    return $this->manyToMany('Role');
}

// in a roles model
public function users()
{
    return $this->manyToMany('User');
}
```

You'll notice that the many to many model is reversible. You can override the adjoining table name, otherwise it will assume that it is the alphabetically ordered names of your models, separated by an underscore. This is why our table it role_user, not user_role.

One Shifts to Many

In addition to the relationships defined above, HUBzero also offers the ability to handle morphing relationships. This introduces the idea of a conditional join, based on an object id and object scope. Say you have a members table that stores membership for both groups and projects. The table structure might look like this:

groups	**projects**	**members**
id	id	id
name	name	scope_id
alias	alias	scope

Our models could then be defined as follows:

```
// in groups model
public function members()
{
    return $this->oneShiftsToMany('Member');
}

// in projects model
public function members()
{
    return $this->oneShiftsToMany('Member');
}
```

Now, instead of joining groups or projects to members by way of projects.id to members.project_id, we associate projects.id to members.scope_id where members.scope is projects.