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.