Agile Database Access with CakePHP 3

Agenda

1. Types of ORMs
2. What I need from ORMs
3. What is agile for me?
4. A simple setup
5. Simple analytical queries
6. More complex examples
7. Query composition and collections
8. Formatting results
9. A last minute tip
10. Debugging Queries
11. Working with JSON
12. Value objects

Types of ORMs

Infuriating ORMs

Types of ORMs

Toy ORMs

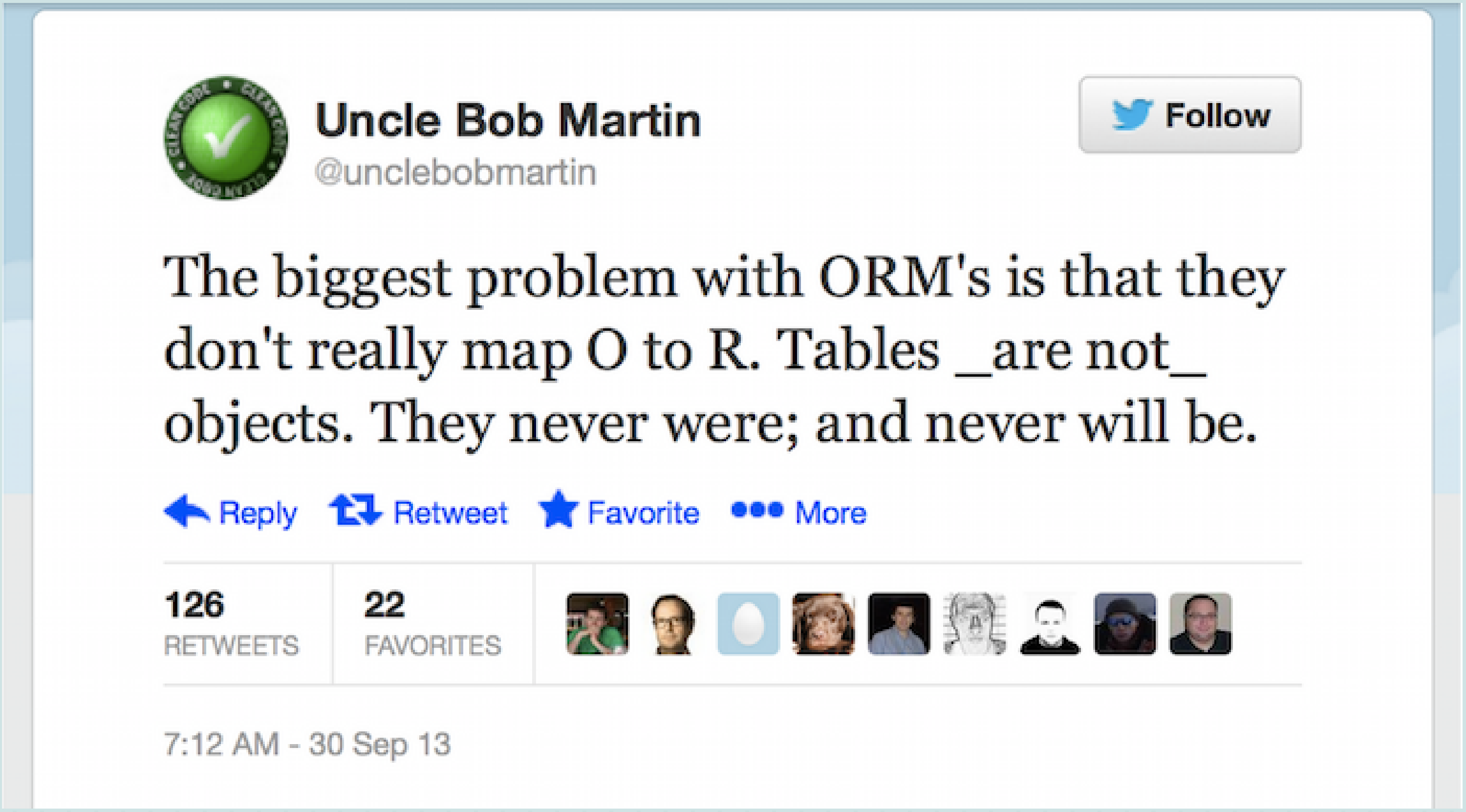
Types of ORMs

Hipster ORMs

Types of ORMs

Awesome ORMs

Some wise words



Uncle Bob Martin
@unclebobmartin

The biggest problem with ORM's is that they don't really map O to R. Tables _are not_ objects. They never were; and never will be.

126 RETWEETS
22 FAVORITES

7:12 AM - 30 Sep 13

What I need from an ORM

- To stop me from repeating the same over and over.
- Help me modularize my common searches.
- Stay out of the way when I want to create complex stuff.
- Testability.
- Ways to hook in and change any default behavior.
- To not hide the Relational aspect of a Relational database.

What is Agile?

- Quick feedback loop.
- Low friction,
- Easy to debug.
- Easy to track.
- Few requirements.
- Ability to scale up.

The Setup

```
class ManagersTable extends Table
{

    public function initialize(array $config = [])
    {
        $this->table('departments_managers');
        $this->primaryKey(['department_id', 'employee_id']);

        $this->belongsTo('Employees', ['joinType' => 'INNER']);
        $this->belongsTo('Departments', ['joinType' => 'INNER']);
    }

    public function beforeFind($event, $query, $options)
    {
        $query->andWhere(['to_date IS' => NULL]);
    }
}
```

The Setup

```
class EmployeesTable extends Table
{

    /**
     * Initialize method
     *
     * @param array $config The configuration for the Table.
     * @return void
     */
    public function initialize(array $config)
    {
        $this->hasMany('Salaries');
        $this->hasMany('Titles');
        $this->belongsToMany('Departments');
    }
}
```

Simple analytical queries

Average historic salary

```php
// In SalariesTable.php
public function findAverage(Query $query, $options = [])
{
    return $query->select(['average' => $query->func()->avg('Salaries.salary')]);
}
```

```json
{
    "average": 63810.74
}
```

Simple analytical queries

Currently hired female managers

```
    public function findFemale(Query $query, $options = [])
    {
        return $query->contain(['Employees'])->where(['Employees.gender' => 'F']);
    }
```

```
SELECT Managers.*, Employees.*
FROM department_managers Managers
INNER JOIN employees Employees ON Employees.id = (Managers.employee_id)
 WHERE Employees.gender = 'F' AND to_date IS NULL
```

A more complex example

Percentage of currently hired female managers

```
public function findFemaleRatio(Query $query, $options = [])
{
    $allManagers = $this->find()->select($query->func()->count('*'));
    $ratio = $query
        ->newExpr($query->func()->count('*'))
        ->type('/')
        ->add($allManagers)
    return $query
        ->find('female')
        ->select(['female_ratio' => $ratio]);
}
```

```
{
    "female_ratio": 0.4444
}
```

Queries can be composed

Average salary of currently hired employees by gender

```
    public function findOfHired(Query $query, $options = [])
    {
        return $query->contain(['Employees'])->where(['Salaries.to_date IS' => null]);
    }

    public function findAveragePerGender(Query $query, $options = [])
    {
        return $query
            ->select(['gender' => 'Employees.gender'])
            ->find('average')
            ->contain(['Employees'])
            ->group(['Employees.gender']);
    }
```

```
$salariesTable
    ->find('ofHired')
    ->find('averagePerGender')
    ->indexBy('gender');
```

Queries are Collections

Yearly salary average per department and gender

```
    public function findAveragePerDepartment(Query $query, $options = [])
    {
        return $query
            ->select(['department' => 'Departments.name'])
            ->find('average')
            ->matching('Employees.Departments')
            ->where([
                'Salaries.from_date < DepartmentsEmployees.to_date',
                'Salaries.from_date >= DepartmentsEmployees.from_date',
            ])
            ->group(['Departments.id']);
    }
```

Queries are Collections

Yearly salary average per department and gender

```
public function findAveragePerYear(Query $query, $options = [])
{
    $year = $query->func()->year(['Salaries.from_date' => 'literal']);
    return $query
        ->select(['year' => $year])
        ->find('average')
        ->group([$year]);
}

$averages = $salariesTable
    ->find('averagePerYear')
    ->find('averagePerDepartment')
    ->find('averagePerGender');
```

Queries are Collections

Yearly salary average per department and gender

```
$averages->groupBy('year')->each(function ($averages, $year) {
    displayYear($year);

    collection($averages)->groupBy('department')->each(function ($d, $averages) {
        displayDepartment($d);
        collection($averages)->each('displayAverage');
    })
});
```

Result Formatters

Pack common post-processing into custom finders

```
public function findGroupedByYearAndDepartment($query)
{
    return $query->formatResults(function ($results) {
        return $results->groupBy('year');
    })
    ->formatResults(function ($years) {
        return $years->map(function ($results) {
            return collection($results)->groupBy('department');
        });
    });
}

$salariesTable
        ->find('averagePerYear')
        ->find('averagePerDepartment')
        ->find('averagePerGender')
        ->find('groupedByYearAndDepartment');
```

Result Formatters

They look sexier in HackLang

```
public function findGroupedByYearAndDepartment($query)
{
    return $query
    ->formatResults($results ==> $results->groupBy('year'))
    ->formatResults($years ==> $years->map(
        $results ==> collection($results)->groupBy('department')
    );
}
```

Associations in another database

Use tables from other databases by specifying the strategy

```
public function initialize(array $config)
{
    $this->hasOne('LinkedEmployee', [
        'className' => 'External\System\EmployeesTable',
        'strategy' => 'select'
    ]);
}
```

- A gotcha: It will not be possible to use `matching()`

Debugging Queries

- `debug($query)` Shows the SQL and bound params, does not show results
- `debug($query->all())` Shows the ResultSet properties (not the results)
- `debug($query->toArray())` An easy way to show each of the results
- `debug(json_encode($query, JSON_PRETTY_PRINT))` More human readable results.
- `debug($query->first())` Show the properties of a single entity.
- `debug((string)$query->first())` Show the properties of a single entity as JSON.

Debugging Queries

Pro tip: create a dj() function

```
function dj($data)
{
    debug(json_encode($data, JSON_PRETTY_PRINT), null, false);
}
```

```
dj($query);
```

```
[
    {
        "average": 0.4444
    }
]
```

Modifying JSON output

I don't want to show primary keys or foreign keys

```
class Employee extends Entity
{
    protected $_hidden = [
        'id'
    ];
}
```

```
class Manager extends Entity
{
    protected $_hidden = [
        'employee_id',
        'department_id'
    ];
}
```

Modifying JSON output

I want to show employees' full name

```
class Employee extends Entity
{
    protected $_virtual = [
        'full_name'
    ];

    protected function _getFullName()
    {
        return $this->name . ' ' . $this->last_name;
    }
}
```

Custom serialization

Let's try to do HAL

```
    public function index()
    {
        $managers = $this->paginate($this->Managers);
        $managers = $managers->map(new LinksEnricher($this->Managers));
        $this->set('managers', $managers);
        $this->set('_serialize', ['managers']);
    }
```

Custom Serialization

Let's try to do HAL

```php
class LinksEnricher
{
...
    public function __invoke(EntityInterface $row)
    {
        $primaryKey = array_values($row->extract((array)$this->table->primaryKey()));
        $row->_links = [
            'self' => [
                'href' => Router::url([
                    'controller' => $row->source(),
                    'action' => 'view',
                ] + $primaryKey)
            ],
        ];
        return $this->enrich($row); // Recurse for associations
    }
...
}
```

```
{
    "managers": [
        {
            "from_date": "1996-01-03T00:00:00+0000",
            "to_date": null,
            "department": {
                "name": "Customer Service",
                "_links": {
                    "self": {
                        "href": "\/departments\/view\/d009"
                    }
                }
            },
            "employee": {
                "birth_date": "1960-03-25T00:00:00+0000",
                "first_name": "Yuchang",
                "last_name": "Weedman",
                "gender": "M",
                "hire_date": "1989-07-10T00:00:00+0000",
                "_links": {
                    "self": {
                        "href": "\/employees\/view\/111939"
                    }
                },
                "full_name": "Yuchang Weedman"
            },
            "_links": {
                "self": {
                    "href": "\/managers\/d009\/111939"
                }
            }
        }
```

Value Objects

Why?

- Allow to add custom logic to dumb data.
- Help with custom serialization
- Make translation and localization easier
- Auto-validation
- Greater integrity.

Value Objects

Adding logic to plain data

```
class Gender implements JsonSerializable
{
    private static $genders = [];

    protected $short;

    protected $name;

    protected function __construct($gender)
    {
        $this->short = $gender;
        $this->name = $gender === 'F' ? 'Female' : 'Male';
    }

    public static function get($gender)
    {
        ...
        return $genders[$gender] = new static($gender);
    }

...
```

Value Objects

Accepting value objects

```php
class Employee extends Entity
{
    protected function _setGender($gender)
    {
        return Gender::get($gender);
    }
```

```php
$employeeEntity->gender = 'F';
get_class($employeeEntity->gender); // App\Model\Value\Gender
$employeeEntity->gender = Gender::get('F');
```

Value Objects

Wiring them to the database

```
class GenderType extends Type
{
...
}
```

```
Type::build('gender', 'App\Model\Database\Type');
```

```
class EmployeesTable extends Table
{
...
    protected function _initializeSchema(Schema $schema)
    {
        $schema->columnType('gender', 'gender');
        return $schema;
    }
}
```

Value Objects

Using them in Queries

```
$employee->gender = Gender::get('F');
$result = $employeesTable->find()->where([['gender' => $employee->gender]])->first();
$employee->gender === $result->gender;
```

- You can use objects as values in where conditions (or any query expression)

Thanks for your time

Questions?

https://github.com/lorenzo/cakephp3-advanced-examples