www.phparch.com

php[architect]

March 2016
VOLUME 15 - ISSUE 3

Mutant Testing

Education Station:
Hunting Mutants with Humbug

Leveling Up:
Ensuring Your Tests are Valuable

ALSO INSIDE

FREE Article!

Object-Relational Mapping with Laravel's Eloquent

Integrating Heterogeneous Web Applications with Laravel and Guzzle

An Introduction to Compilers, Interpreters, and JITs

Getting Started with Zend Expressive

Community Corner: Modern PHP

finally{}: Growing Programmer Population

Security Corner: Securing Legacy Applications—Part 1

Object-Relational Mapping with Laravel's Eloquent

Luis Atencio

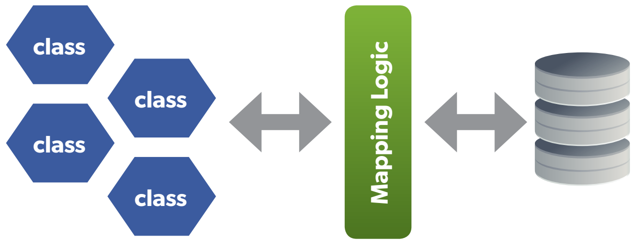
Object-relational mapping—ORM for short—has been a challenge in software system development for many years. In basic terms, this difficulty arises when attempting to match an object-oriented system to a relational database or RDBMS. Objects are very free and lenient structures that can contain many different types of properties, including other objects. While object graphs are free to grow organically via inheritance, relational entities are rather flat and contrived, defining a very limited set of types. So can you transpose an object-oriented model onto a relational model? While this remains a very hard problem to solve, there are ORM solutions that can emulate the same effect by creating a virtual object database. One such solution is Laravel's Eloquent ORM framework. Eloquent provides the abstractions needed for you to be able to work with relational data as if it were loaded onto an inherited object model.

Introduction

ORM Mapping is a technique for converting data from the world of objects into the world of relations (and vice versa), or tables in the sense of a typical RDBMS. This eliminates the need for building a very complex adapter layer in charge of reading relational data from, say, a MySQL database, into objects. ORM tools also abstract out the details of mapping logic, i.e., managing reads and writes, as well as one-to-one or one-to-many relationships.

In case you're not familiar with the technology, I'll provide brief introductions to both Laravel and Eloquent.

ORM Mapping **FIGURE 1**



Laravel

Laravel[1] is a PHP web MVC application framework designed to abstract the common pain points associated with applications pertaining to: authentication, routing, middleware, caching, unit testing, and inversion of control. In many ways, it's similar to the Rails platform that dominates Ruby web development. Built into Laravel is a component called Eloquent ORM.

1 *Laravel: https://laravel.com*

Eloquent ORM

Eloquent ORM[2] is PHP's response to very successful ORM frameworks such as Hibernate and Rails, available in Java and Ruby respectively, for many years. These ORM tools nicely implement the *ActiveRecord*[3] design pattern that treats objects as relational rows in a database. This pattern puts the **M** in MVC—the model—which facilitates the creation of objects whose data need to be persisted and read from a database. In simple terms, in *ActiveRecord* the bulk of the logic for carrying out data access operations are shoved into your model classes, in contrast to having it all reside in a data access object (DAO). In this model, the class definition maps to the relational table per se, while instances of the object constitute the individual rows.

Common ORM Strategies

ORMs, like Eloquent, connect these rich objects to tables using different strategies. In this article, I will go over the two most popular ones:

- Concrete Table Strategy[4]
- Single Table Strategy[5]

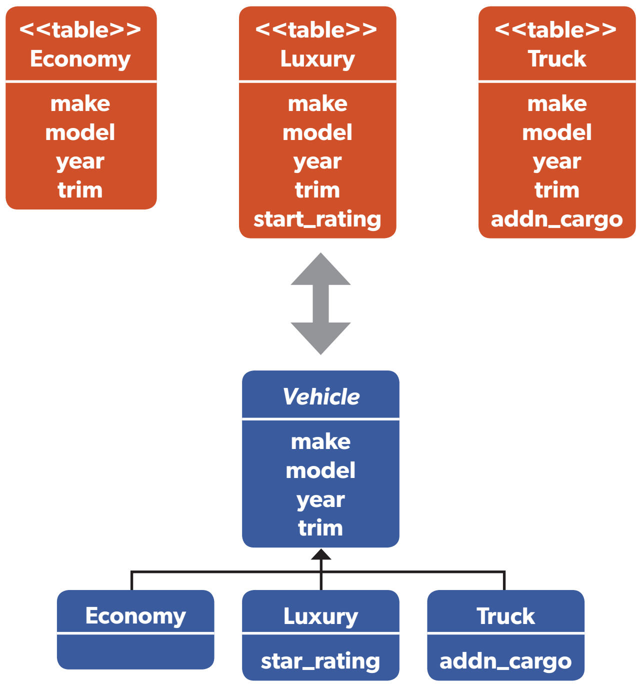
Concrete (Class) Table Inheritance (CTI)

Because relational databases do not support inheritance (theoretically speaking), thinking of tables from an object instance point of view is incredibly challenging. Given that there is no automatic way for data to trickle down from "parent" tables to any "derived" tables, it's natural to think that each child object would map to its

2 *Eloquent ORM: https://laravel.com/docs/5.0/eloquent*
3 *Active Record Patter: http://phpa.me/fowler-ar*
4 *Concrete Table Inheritance: http://phpa.me/fowler-sti*
5 *Single Table Inheritance: http://phpa.me/fowler-sti*

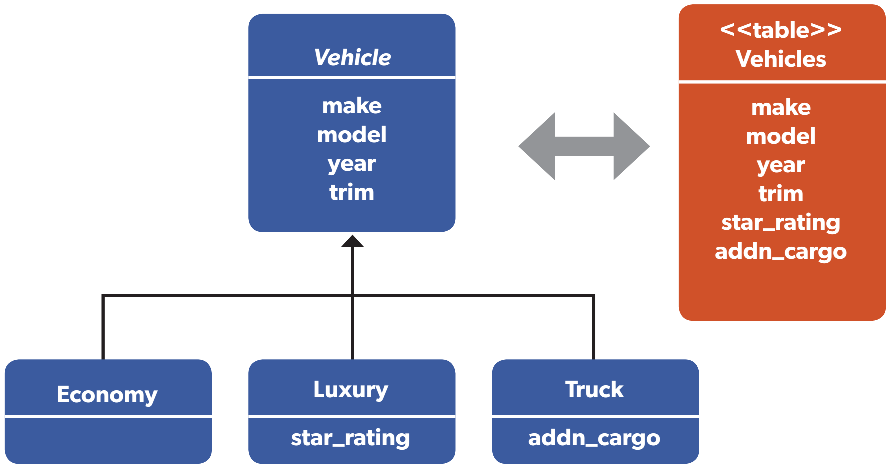
own concrete table. Consider this very simple model for a car dealership application:

Concrete Mapping **FIGURE 2**



And this strategy would work well for very diverse classes or data types that share only a minimal set of attributes and contain many specialized child attributes, but results in lots of duplication when the amount of inherited data is greater. As in this case, you can see that I needed to repeat the *make*, *model*, *year*, and *trim* columns in every table. It seems that for this example, since we have very simple child types and flat class hierarchy, I could benefit from consolidating the data of an object model into a single table, known as Single Table Inheritance.

Single-Table Mapping **FIGURE 3**



Single Table Inheritance (STI)

Instead of creating dedicated tables for each child type, a single table is used to house all of the data contained inside an object graph. A *type* column, also known as a *Discriminator* column, is used to discern among the different types of objects in the graph.

Despite storing the data in a single table, ORM tools can easily map this onto different objects at runtime, abstracting the persistence strategy from you and the application. Let's jump into the implementation details.

Implementation

Before we get deep into the object definitions and queries, let's take care of some housekeeping application setup. Because Laravel and all of its packages are loaded via Composer, let's begin by defining the project level `composer.json` file. I will be using Laravel 5.0, see Listing 1.

LISTING 1

```
{
    "name": "phparch/eloquent-laravel-demo",
    "description": "PHPArch Laravel Eloquent Sample Code",
    "type": "project",
    "require": {
        "laravel/framework": "5.0.20",
        "phaza/single-table-inheritance": "1.0.1"
    },
    "require-dev": {
        "phpunit/phpunit": "4.6.2",
        "phpspec/phpspec": "~2.1"
    },
    "autoload": {
        "classmap": [
            "database"
        ],
        "psr-4": {
            "App\\": "app/"
        }
    }
}
```

Configuration

In Listing 1, I left out some of the other configuration parameters and kept the pertinent ones. If you've set up a Laravel application before, you should be familiar with this configuration. As you know, Laravel sets up the web application code under the `app` folder, which corresponds to the `App` namespace. Here you will find all of the code related to controllers, events, handlers, routing, etc.

Instead of sprinkling custom code everywhere, I will create a project directory for all of my custom classes called `EloquentDemo`.

Class Structure

The class structure (model) that drives all of the business logic and persistence will be stored in the `Model` directory, defining the package `App\EloquentDemo\Model`, underneath which my virtual object database will reside.

BaseModel

Every object that will be acting as an *ActiveRecord* must be an instance of Eloquent's `Model`. `Model` defines abstract behavior that applies generally to all instances. Because there's some general behavior that applies to all of my application objects as well, it's a good idea to create an abstract class between my custom classes and Eloquent's `Model` class. This will give you additional flexibility in the future to define shared properties and methods. Suppose I want to use soft deletes instead of hard deletes—I can easily install this trait in the base class (See Listing 2).

LISTING 2

```php
<?php
namespace App\EloquentDemo\Model;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;

/**
 * Base class for all models in the system
 *
 * @author Luis Atencio
 * @package App\EloquentDemo\Model
 */
abstract class BaseModel extends Model {

   use SoftDeletes;

   /**
    * Return the database record ID for all tables
    *
    * @return integer
    */
   public function getId() {
      return $this->id;
   }

   /**
    * Save the record ID
    *
    * @return mixed
    */
   public function setId($id) {
      $this->id = $id;
   }
}
```

Parent Type = Single Table

With this out of the way, I will create the object hierarchy that takes advantage of single inheritance. I will define an abstract `Vehicle` class, which corresponds to the `vehicles` database table, which I'll create with the Laravel migration script in Listing 3.

To implement the object-table mapping, I will use the **phaza/single-table-inheritance**[6] composer package (configuration shown in Listing 4) to instrument the parent type, via a trait.

This library requires a very minimal set of artifacts: a protected `$singleTableTypeField` field used to define the discriminator column and `$singleTableSubclasses` array containing the path to the model classes whose data will be shared by this single table. This is really clean because you don't have to expose the internal details of the inheritance mapping to Controller layer, allowing the trait to internally take complete control of these objects and their behavior.

LISTING 3

```php
<?php
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateVehicleTable extends Migration {

   /**
    * Create vehicle table
    *
    * @return void
    */
   public function up() {
      Schema::create('vehicles', function(Blueprint $table) {
         $table->increments('id');
         $table->string('vehicle_type');
         $table->unsignedInteger('make_id');
         $table->unsignedInteger('model_id');
         $table->unsignedInteger('trim_id');
         $table->unsignedInteger('year');
         $table->boolean('addn_cargo')->default(false);
         $table->tinyInteger('star_rating')->nullable();
         $table->timestamps();
         $table->softDeletes();
      });
   }

   /**
    * Drop vehicle table
    *
    * @return void
    */
   public function down() {
      Schema::drop('vehicles');
   }
}
```

6 *phaza/single-table-inheritance: http://phpa.me/phaza-sti*

LISTING 4

```php
<?php

namespace App\EloquentDemo\Model;

use DB;
use Phaza\SingleTableInheritance\SingleTableInheritanceTrait;

/**
 * Represents parent base model
 *
 * @author luisat
 * @package App\EloquentDemo\Model
 */
class Vehicle extends BaseModel {

   use SingleTableInheritanceTrait;

   /**
    * The database table used by the model.
    *
    * @var string
    */
   protected $table = 'vehicles';

   /**
    * STI table column name
    *
    * @var string
    */
   protected static $singleTableTypeField = 'vehicle_type';

   /**
    * Display name
    *
    * @var string
    */
   protected $displayName;

   /**
    * STI class names
    *
    * @var array
    */
   protected static $singleTableSubclasses = [
      Economy::class, Truck::class, Luxury::class
   ];

   /**
   * The attributes that are mass assignable when reading from DB
    *
    * @var array
    */
   protected $fillable = [
      'vehicle_type',
      'make_id',
      'model_id',
      'trim_id',
      'year',
      'addn_cargo',
      'star_rating'
   ];

   public function getType() {
      return $this->vehicle_type;
   }

   public function getMakeId() {
      return $this->make_id;
   }

   public function getModelId() {
      return $this->model_id;
   }

   public function getTrimId() {
      return $this->trim_id;
   }

   /**
    * Load the Make record for this vehicle
    * @return Make
    */
   public function make() {
      return $this->hasOne(Make::class, 'id', 'make_id');
   }

   /**
    * Load the Model record for this vehicle
    * @return Model
    */
   public function model() {
      return $this->hasOne(Model::class, 'id', 'model_id');
   }

   /**
    * Load the Trim record for this vehicle
    * @return Trim
    */
   public function trim() {
      return $this->hasOne(Trim::class, 'id', 'trim_id');
   }
}
```

Now, let's create the subtypes. To keep things short, I'll just show one of the child objects, `Truck` (see Listing 5); you can decipher the rest easily.

Now that we've set this up, let's show how easy it is to create and read records of any type.

Writing to the Database

I can write a record in the database explicitly by defining type (see Listing 6).

Better yet, I can leverage the functionality of STI to create and read items of any type as shown in Listing 7.

Reading from the Database

This single table strategy allows you to read data polymorphically for any vehicle type, and seamlessly taking care of any associations just like any other model class would. Consider the simple queries in Listing 8.

As you can see, by using Laravel's Eloquent ORM with this configuration there's absolutely nothing else you need to do to support STI. This is the real beauty of a setup like this. In an MVC world, you make your changes at the database and Model levels but your Controllers and Views are handled just like any other model. As far as you are concerned, each object has its own persistent storage.

Pros and Cons

There are many considerations to keep in mind when deciding whether STI is suitable for your application. Here's a short list of the pros and cons of using STI:

LISTING 5

```
<?php namespace App\EloquentDemo\Model;

/**
 * Derived vehicle type: Truck
 *
 * @author luisat
 * @package App\EloquentDemo\Model
 */
class Truck extends Vehicle
{
   /**
    * Single inheritance table type
    *
    * @var string
    */
   protected static $singleTableType = 'Truck';

   /**
    * Display name for the model
    *
    * @var string
    */
   protected $displayName = 'Truck';


   public function getAdditionalCargo() {
      return $this->addn_cargo;
   }

   public function setAdditionalCargo($addnCargo) {
      $this->addn_cargo = $addnCargo;
   }
}
```

LISTING 6

```
$data = [
  'vehicle_type' => 'Luxury',
  'make_id'      => Make::firstByAttributes(['name' => 'Toyota'])->id,
  'model_id'     => Model::firstByAttributes(['name'=> 'Camry'])->id,
  'trim_id'      => Trim::firstByAttributes(['name' => 'Camry LE'])->id,
  'year'         => 2015,
  'addn_cargo'   => false,
  'star_rating'  => 4
];

$camry = Vehicle::create($data);

//...

Vehicle::find($camry->id); //-> Vehicle(1)
```

LISTING 7

```
Truck::create([
    'make_id'  => Make::firstByAttributes(['name' => 'Ford'])->id,
    'model_id' => Model::firstByAttributes(['name'=> 'F150'])->id,
    'trim_id'  => Trim::firstByAttributes(['name' => 'Limited'])->id,
    'year' => 2015,
    'addn_cargo' => true,
    'star_rating' => 5
]);
```

LISTING 8

```
// Reading and updating a collection of truck records
Truck::where('year', '>', 2014)
       ->update(['trim_id' => Trim::firstByAttributes(
              ['name' => 'XL'])->id]
          );

// Load associations
Luxury::find($camry->id)->model; //-> Model('Camry')
Luxury::find($camry->id)->make;  //-> Make('Toyota')

// Query all vehicles in the database
Vehicle::all(); //-> Collection[Luxury(1), Truck(2), ...]
```

Pros

- Simple to implement, perhaps for quick spiking tasks and simple apps.
- Easy to add new classes by simply adding additional columns.
- Supports polymorphic queries with a simple discriminator column.
- Data access and reporting are quick since all the information is stored in one table.

Cons

- Tight coupling of objects to tables. A major refactoring effort on your classes would cause changes to your table structure as well.
- Space potentially wasted due to jagged data. This can occur when child types evolve with many unique attributes.
- Table quickly grows when supporting deeply nested hierarchies.

When to use

This is ideal for simple and/or shallow class hierarchies that have lots of overlap with parent types and clear inheritance hierarchy.

Final Comments

In this article I showed you how to use a Single Table Inheritance (STI) mapping scheme to model a simple object graph. This will allow you to emulate inheritance in relational databases. The caveat to this strategy, though, is that there must be a clear and natural class hierarchy; otherwise, STI may be hard to maintain. A common problem that might occur is that your child types develop too many unique attributes, creating lots of non-global columns in a single table.

ORM frameworks like Laravel's Eloquent can be used to work with several strategies very effectively, but ORMs are not the only solution. Other people gravitate toward the NoSQL databases, such as MongoDB, because it allows them more flexibility as data is stored more freely in a schemaless object form. This avoids having to translate between the contrived, restricted relational form into objects. Nevertheless the strong principles behind relational databases continue to be very appealing for developers and system architects, so ORMs are and will be used very frequently in modern web applications for many years to come.

Luis Atencio is a Staff Software Engineer for Citrix Systems in Ft. Lauderdale, FL. He has a B.S. and an M.S. in Computer Science. He works full time developing and architecting web applications with Java, PHP, and JavaScript. When he is not coding, Luis writes a developer blog at http://luisatencio.net focused on software engineering. Luis is also the author of Functional Programming in JavaScript[7] (Manning 2016). @luijar

7 https://www.manning.com/books/functional-programming-in-javascript