Laravel Design Patterns and Best Practices

Arda Kılıçdağıe
H. İbrahim YILMAZ

Community Experience Distilled
Laravel Design Patterns and Best Practices
Enhance the quality of your web applications by efficiently implementing design patterns in Laravel
Arda Kılıçdağı
H. İbrahim YILMAZ
PACKT open source

Chapter No. 2
"Models in MVC"

In this package, you will find:

The authors biography

A preview chapter from the book, Chapter no.2 "Models in MVC"

A synopsis of the book’s content

Information on where to buy this book

About the Authors

Arda Kılıçdağı is a PHP/MySQL/JavaScript/Linux programmer and server administrator from Turkey. He has been developing applications with PHP since2005. He administers the Turkish national support site of the world famous open source content management script, PHP-Fusion. He's also one of the international developers and a member of the management team of CMS, and he plays an important role in CMS's future. He has worked as a developer and has experience on projects such as Begendy (an exclusive private shopping website) and Futbolkurdu (a local soccer news website). He has experience working with the Facebook API, the Twitter API, and PayPal's Adaptive Payment API (used on crowdfunding websites such as KickStarter). He's also experienced with JavaScript, and he's infusing his applications with JavaScript and jQuery, both on frontend and backend sides.

He has developed applications using CodeIgniter and CakePHP for about 4 years, but these PHP frameworks didn't suit his needs completely, and that's why he decided to use another framework for his projects. After getting introduced to Laravel, he has developed all his applications with it.

He's also interested in Unix and Linux, and he uses Linux on a daily basis. He's administering the world's best-known microcomputer, Raspberry Pi's biggest Turkish community website, Raspberry Pi Türkiye Topluluğu (Raspberry Pi Turkish Community Website).

Before authoring this book, Arda has written two other books. The first book is *Laravel Application Development Blueprints*, *Packt Publishing*, coauthored by H. İbrahim YILMAZ. The second book, *Raspberry Pi, Dikeyeksen Consulting & Publishing*, is written in Turkish.

H. İbrahim YILMAZ is a daddy, developer, geek, and an e-commerce consultant from Turkey. After his education at Münster University, Germany, he worked as a developer and software coordinator in over a dozen ventures. During this period, he developed the usage of APIs such as Google, YouTube, Facebook, Twitter, Grooveshark, and PayPal.

Currently, he's focused on creating his company about concurrent computing, Big Data, and game programming. He writes articles on Erlang, Riak, functional programming, and Big Data on his personal blog at `http://blog.drlinux.org` He is a big Pink Floyd fan, playing bass guitar is his hobby, and he writes poems at `http://okyan.us`.

He has a daughter called İklim. He lives in a house full of Linux boxes in Istanbul, Turkey.

I'd like to thank my daughter İklim and my family for their presence. I'd also like to thank the Gezi Park protesters for their cause to make the world a better place.

I'd like to dedicate this book to Berkin Elvan. Berkin was a 15-year-old boy who was hit on the head by a teargas canister fi red by a police officer in Istanbul, while out to buy bread for his family during the June 2013 antigovernment protests in Turkey. He died on March 11, 2014, following a 269-day coma.

Laravel Design Patterns and Best Practices

This book covers how to develop different applications and solve recurring problems using Laravel 4 design patterns. It will walk you through the widely used design patterns—the Builder (Manager) pattern, the Factory pattern, the Repository pattern, and the Strategy pattern—and will empower you to use these patterns while developing various applications with Laravel. This book will help you find stable and acceptable solutions, thereby improving the quality of your applications. Throughout the course of the book, you will be introduced to a number of clear, practical examples about PHP design patterns and their usage in various projects. You will also get acquainted with the best practices for Laravel, which will greatly reduce the probability of introducing errors into your web applications.

By the end of this book, you will be accustomed with Laravel best practices and the important design patterns to make a great website.

What This Book Covers

Chapter 1, Design and Architectural Pattern Fundamentals, explains design and architectural pattern terms and explains the classification of these design patterns and their elements. This chapter provides some examples from the Laravel core code, which contains the design patterns used in the framework. At end of this chapter, the Mode-View-Controller (MVC) architectural pattern and its benefits will be explained.

Chapter 2, Models in MVC, covers the function of the Model layer in the MVC architectural pattern, its structure, its purpose, its role in the SOLID design pattern, how Laravel uses it, and the advantages of Laravel's Model layers and Eloquent ORM. Laravel classes that handle data are also discussed.

Chapter 3, Views in MVC, covers the function of the View layer in the MVC architectural pattern, its structure, its purpose, and the advantages of Laravel's View layer and Blade template engine. The role of View in the MVC pattern and Laravel's approach to that is also covered.

Chapter 4, Controllers in MVC, covers the function of the Controller layer in the MVC architectural pattern, its structure, its purpose, and its usage in Laravel's structure.

Chapter 5, Design Patterns in Laravel, discusses the design patterns used in Laravel. We will also see how and why they are used, with examples.

Chapter 6, Best Practices in Laravel, will cover basic and advanced practices in Laravel, examples of design patterns used in Laravel that we were described in previous chapters, and the reasons these patterns are used

2
Models in MVC

Throughout the chapter, we will be discussing what Model is in the MVC structure, what its purpose is, what its role is in the SOLID design pattern, how Laravel defines it, and the advantages of Laravel's Model layers and Eloquent ORM. We will also discuss Laravel's classes related to handling data.

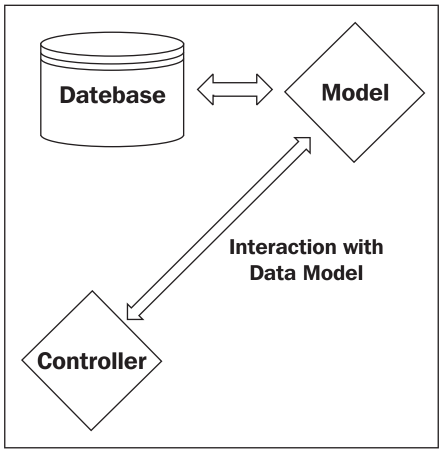
The following is the list of topics that will be covered in this chapter:

- The meaning of the Model
- The roles of the Model in a solid MVC design pattern
- The Model and Model Instances
- How Laravel defines the Model
- The database-related classes of Laravel

What is a Model?

The Model is that part of the Model-View-Controller design pattern that we can simply describe as the layer of the design pattern that handles the management of the data, which is received from the corresponding layers and then sent back to those layers. One thing to note here is that the Model does not know where the data comes from and how it is received.

In simple words, we can say that the Model implements the business logic of the application. The Model is responsible for fetching the data and converting it into more meaningful data that can be managed by other layers of the application and sending it back to corresponding layers. The Model is another name for the domain layer or business layer of an application.



Purposes of the Model

The Model in an application manages all of the dynamic data (anything that's not hardcoded and comes from a database driver) and lets other related components of the application know about the changes. For example, let's say that there is a news article in your database. If you alter it from the database manually, when a route is called—and due to this request—the Controller requests for the data over the Model after the request from the Routing handler, and the Controller receives the updated data from the Model. As a result, it sends this updated data to the View, and the end user sees the changes from the response. All of these data-related interactions are the tasks of the Model. This "data" that the Model handles does not always have to be database related. In some implementations, the Model can also be used to handle some temporary session variables.

The basic purposes of the Model in a general MVC pattern are as follows:

- To fetch the data using a specified (database) driver
- To validate the data
- To store the data
- To update the data

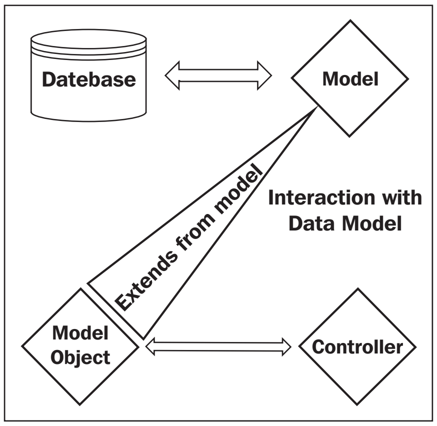
- To delete the data
- To create conditional relations
- To monitor the file I/O
- To interact with third-party web services
- To handle caches and sessions

As you can see, if you follow with an MVC pattern consistently, the Model covers a huge percentage of the application's logic. In modern frameworks, there is a common mistake about the Model that is made by users when learning design patterns. They usually confuse the Model with Model Instances. Although they are quite similar, they have different meanings.

Model instances

In your application, there will usually be more than one data structure to manage. For example, let's say you are running a blog. In a simple blog system, there are authors, blog posts, tags, and comments. Let's say you want to update a blog post; how do you define that the data you want to update is for the blog posts? This is where Model Instances come handy.

Model Instances are simple classes that mostly extend from the Model layer of the application. These instances separate the data logic for each section of your application. In our example, we have four sections to handle (users, posts, tags, and comments). If we are going to handle these using the Model, we have to create at least four instances (we will cover why it is at least four and not exactly four in the *Relationships* section that is under *Eloquent ORM* in this chapter).



As you can see from the diagram, the Controller interacts with the Model Instance to fetch data. Since Model Instances extend from the Model itself, instead of raw Model output, the Controller can customize it or add other layers (such as Validation) to the process.

Let's say you want to fetch the user who has the username `George`. If no Validation layer has been added from the Model Instance to the database, the `username` parameter will go to the database directly, which could be harmful. If you did add a validation layer on your Model Instance (which checks if the username parameter is a clean string), even if the parameter was SQL injection code, it would first be filtered by the validation layer instead of going to the database directly and then be detected as harmful code. Then, the Model Instance will return a message or an exception to the Controller that the parameter is invalid. Then, the Controller will send a corresponding message to the View, and from there, the message will be shown to the end user. In this process, optionally, the application might even fire an Event to log this attempt.

The Model in Laravel

If you recall, we mentioned earlier in this chapter that there are many important jobs that the Model needs to handle. Laravel 4 does not use the MVC pattern directly, but it extends the pattern further. For example, Validation—which is part of the Model in the solid MVC pattern—has its own class, but it's not part of the Model itself. The database connection layer has its own classes for each database driver, but they are not packed in the Model directly. This brings testability, modularity, and extensibility to the Models.

The Laravel Model structure focuses more on the database processes directly, and it is separated from other purposes. The other purposes are categorized as Facades.

To access the database, Laravel has two classes. The first one is Fluent Query Builder and the second one is Eloquent ORM.

Fluent Query Builder

Fluent is the query builder class of Laravel 4. Fluent Query Builder handles base database query actions using PHP Data Objects in the backend, and it can be used with almost any database driver. Let's say that you need to change the database driver from SQLite to MySQL; if you've written your queries using Fluent, then you mostly don't need to rewrite or alter your code unless you've written raw queries using `DB::raw()`. Fluent handles this behind the scenes.

Let's take a quick look at the Eloquent Model of Laravel 4 (which can be found in the `Vendor\Laravel\Framework\src\Illuminate\Database\Query` folder):

```
<?php namespace Illuminate\Database\Query;

use Closure;
use Illuminate\Support\Collection;
use Illuminate\Database\ConnectionInterface;
use Illuminate\Database\Query\Grammars\Grammar;
use Illuminate\Database\Query\Processors\Processor;

class Builder {
    //methods and variables come here
}
```

As you can see, the Eloquent Model uses some classes, such as `Database`, `ConnectionInterface`, `Collection`, `Grammar`, and `Processor`. All of these are required to standardize database queries in the backend, cache the queries if required, and return the output as a collection object.

The following are some basic examples that present how the queries look:

- To get all of the names and show them one by one from a `users` table, use the following code:

  ```
  $users = DB::table('users')->get();
  foreach ($users as $user)
  {
      var_dump($user->name);
  }
  ```

  The `get()` method fetches all of the records from the table in the form of a collection. With a `foreach()` loop, the records are looped, and then we access each name column using `->name` (an object). If the column we want to access is an e-mail, then it'll look like `$user->email`.

- To fetch the first user named `Arda` from the `users` table, use the following code:

  ```
  $user = DB::table('users')->where('name', 'Arda')->first();
  var_dump($user->name);
  ```

  The `where()` method filters the query with given parameters. The `first()` method directly returns the collection object of a single item from the first matched element. If there were two users named `Arda`, only the first one would be caught and set to the `$user` variable.

- If you wanted to use `OR` statements in the `where` clauses, you could use the following code:

  ```
  $user = DB::table('users')
  ->where('name', 'Arda')
  ->orWhere('name', 'Ibrahim')
  ->first();
  var_dump($user->name);
  ```

- To use operators in the `where` clauses, the following third parameter should be added between the column name and variable that is to be filtered:

  ```
  $user = DB::table('users')->where('id', '>', '2')->get();
  foreach ($users as $user)
  {
      var_dump($user->email);
  }
  ```

- If you are using offsets and limits, execute the following query:

  ```
  $users = DB::table('users')->skip(10)->take(5)->get();
  ```

  This produces `SELECT * FROM` users `LIMIT 10,5` in MySQL. The `skip($integer)` method will set an offset to the query, and `take($ integer)` will limit the output by the natural number that has been set as the parameter.

- You can also limit what is to be fetched using the `select()` method and use the following `join` statements easily in Fluent Query Builder:

  ```
  DB::table('users')
     ->join('contacts', 'users.id', '=', 'contacts.user_id')
     ->join('orders', 'users.id', '=', 'orders.user_id')
     ->select('users.id', 'contacts.phone', 'orders.price');
  ```

  These methods simply join the `users` table with contacts, then join orders with users, and then get the user ID, the phone column from the `contacts` table, and the price column from the `orders` table.

- You can group queries by parameters easily, using closure functions. This will allow you to write more complicated queries with ease, as follows:

  ```
  DB::table('users')
      ->where('name', '=', 'John')
      ->orWhere(function($query)
      {
          $query->where('votes', '>', 100)
                ->where('title', '<>', 'Admin');
  ```

```
    })
    ->get();
```

This will produce the following SQL query:

```
select * from users
   where name = 'John'
   or
   (votes > 100 and title <> 'Admin')
```

- You can also use aggregations in the query builder (such as `count`, `max`, `min`, `avg`, and `sum`) as follows:

```
$users = DB::table('users')->count();
$price = DB::table('orders')->max('price');
```

- Sometimes, such builders might not be enough, or you might want to run raw queries. You can also wrap your raw queries inside of Fluent as follows:

```
$users = DB::table('users')
     ->select(
array(
DB::raw('count(*) as user_count'),
'status',
)
)
     ->where('status', '<>', 1)
     ->groupBy('status')
     ->get();
```

- To insert new data into the table, use the `insert()` method:

```
DB::table('users')->insert(
    array('email' => 'me@ardakilicdagi.com', 'points' => 100)
);
```

- To update a row(s) from a table, use the `update()` method:

```
DB::table('users')
->where('id', 1)
->update(array('votes' => 100));
```

- To delete a row(s) from a table, use the `delete()` method:

```
DB::table('users')
->where('last_login', '2013-01-01 00:00:00')
->delete();
```

- Benefiting of `CachingIterator` that's used the `Collection` class, Fluent Query Builder can also cache results upon calling using the method `remember()`:

  ```
  $user = DB::table('users')
  ->where('name', 'Arda')
  ->remember(10)
  ->first();
  ```

  After this query is called once, it's cached for 10 minutes; if this query is called again, instead of fetching from the database, it'll fetch directly from the cache instead, until 10 minutes have passed.

Eloquent ORM

Eloquent ORM is the Active Record implementation in Laravel. It's simple, powerful, and easy to handle and manage.

For each database table, you'll need a new Model Instance to benefit from Eloquent.

Let's say you have a `posts` table, and you want to benefit from Eloquent; you need to navigate to `app/models` and save this file as `Post.php` (the singular form of the table name):

```
<?php class Post extends Eloquent {}
```

And that's it! You're ready to benefit from Eloquent methods for your table.

Laravel allows you to assign any table to any Eloquent Model Instance. It's not necessary, but it's a good habit to name Model Instances with the singular name of the corresponding table. This name should be a singular form of the table name it represents. If you have to use a name that does not follow this general rule, you can do so by setting a protected `$table` variable inside of the Model Instance.

```
<?php Class Post Extends Eloquent {

   protected $table = 'my_uber_posts_table';

}
```

This way, you can assign a table to any desired Model Instance.

It's not necessary to add the instance into the `models` folder in `app`. As long as you've set an `autoload` path in `composer.json`, you can get rid of this folder completely and add it wherever you like. This will bring flexibility to your architecture during programming.

Let's take a quick look at the following `Model` class of Laravel 4 that we just extended from (which is in the `Vendor\Laravel\Framework\src\Illuminate\Database\Eloquent` folder):

```
<?php namespace Illuminate\Database\Eloquent;

use DateTime;
use ArrayAccess;
use Carbon\Carbon;
use LogicException;
use JsonSerializable;
use Illuminate\Events\Dispatcher;
use Illuminate\Database\Eloquent\Relations\Pivot;
use Illuminate\Database\Eloquent\Relations\HasOne;
use Illuminate\Database\Eloquent\Relations\HasMany;
use Illuminate\Support\Contracts\JsonableInterface;
use Illuminate\Support\Contracts\ArrayableInterface;
use Illuminate\Database\Eloquent\Relations\Relation;
use Illuminate\Database\Eloquent\Relations\MorphOne;
use Illuminate\Database\Eloquent\Relations\MorphMany;
use Illuminate\Database\Eloquent\Relations\BelongsTo;
use Illuminate\Database\Query\Builder as QueryBuilder;
use Illuminate\Database\Eloquent\Relations\MorphToMany;
use Illuminate\Database\Eloquent\Relations\BelongsToMany;
use Illuminate\Database\Eloquent\Relations\HasManyThrough;
use Illuminate\Database\ConnectionResolverInterface as Resolver;

abstract class Model implements ArrayAccess, ArrayableInterface,
JsonableInterface, JsonSerializable {
   //Methods and variables come here
}
```

Eloquent uses `Illuminate\Database\Query\Builder`, which is the Fluent Query Builder that we described earlier, and its methods are defined inside it. Thanks to this, all of the methods that can be defined in Fluent Query Builder can also be used in Eloquent ORM.

As you can see, all of the used classes are split according to their purpose. This brings a better **Abstraction** and **Reusability** to the architecture.

Relationships

Eloquent ORM has other benefits in addition to Fluent Query Builder. The major benefit is Model Instance Relations, which allows Fluent Query Builder to form a relationship with other Model Instances easily. Let's say you have `users` and `posts` tables, and you want to get the posts made by a user with an ID of `5`. After the relationship is set, the collection of these posts can be fetched with this code easily:

```
User::find(5)->posts;
```

This couldn't be easier, could it? There are three major relationship types: one-to-one, one-to-many, and many-to-many. In addition to these, Laravel 4 also has the has-many-through and morph-to-many (many-to-many polymorphic) relationships:

- **One-to-one relationships**: These are used when both Models have only one element of each other. Let's say you have a `User` Model, which should only have one element in your `Phone` Model. In this case, the relationship will be defined as follows:

  ```
  //User.php model
  Class User Extends Eloquent {

     public function phone() {
        return $this->hasOne('Phone'); //Phone is the name of
          Model Instance here, not a table or column name
     }

  }

  //Phone.php model
  Class Phone Extends Eloquent {

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

  }
  ```

- **One-to-many relationships**: These are used when a Model has more than one element of another. Let's say you have a news system with categories. A category can have more than one item. In this case, the relationship will be defined as follows:

  ```
  //Category.php model
  class Category extends Eloquent {

     public function news() {
  ```

```
      return $this->hasMany('News'); //News is the name of
        Model Instance here
   }

}

//News.php model
class News extends Eloquent {

   public function categories() {
      return $this->belongsTo('Category');
   }

}
```

- **Many-to-many relationships**: These are used when two Models have more than one element of each other. Let's say you have `Blog` and `Tag` Models. A blog post might have more than one tag, and a tag might be assigned to more than one blog post. For such instances, a pivot table is used along with Many to Many Relationships. The relationship can be defined as follows:

```
//Blog.php Model
Class Blog Extends Eloquent {

   public function tags() {
      return $this->belongsToMany('Tag', 'blog_tag');
        //blog_tag is the name of the pivot table
   }

}

//Tag.php model
Class Tag Extends Eloquent {

   public function blogs() {
      return $this->belongsToMany('Blog', 'blog_tag');
   }

}
```

Laravel 4 adds some flexibility and additional relationships to these known relationships. They are "has-many-through" and "polymorphic relationships".

- **Has-many-through relationships**: These are more like shortcuts. Let's say you have a `Country` Model, `User` Model, and `Post` Model. A country may have more than one user, and a user may have more than one post. If you want to access all of the posts created by the users of a specific country, you need to define the relationship as follows:

```
//Country.php Model
Class Country Extends Eloquent {

   public function posts() {
      return $this->hasManyThrough('Post', 'User');
   }

}
```

- **Polymorphic relationships**: These are featured in Laravel v4.1. Let's say you have a `News` Model, `Blog` Model, and `Photo` Model. This `Photo` Model holds images for both `News` and `Blog`, but how do you relate this or identify a specific photo that is either for blogs or posts? This can be done easily. It needs to be set as follows:

```
//Photo.php Model
Class Photo Extends Eloquent {

   public function imageable() {
      return $this->morphTo(); //This method doesn't take
       any parameters, Eloquent will understand what will
         be morphed by calling this method
   }

}

//News.php Model
Class News Extends Eloquent {

   public function photos() {
      return $this->morphMany('Photo', 'imageable');
   }

}

//Blog.php Model
Class Blog Extends Eloquent {

   public function photos() {
      return $this->morphMany('Photo', 'imageable');
   }

}
```

The keyword imageable, which will describe the owner of the image, is not a must; it could be anything, but you need to set it as a method name and put it as a second parameter into morphMany relationship definitions. This helps us understand how we're going to access the owner of the photo. This way, we can call this easily from the Photo Model without needing to understand whether its owner is Blog or News:

```
$photo = Photo::find(1);
$owner = $photo->imageable; //This brings either a blog
  collection or News according to its owner.
```

In this example, you'll need to add two additional columns to your Photo Model's table. These columns are imageable_id and imageable_type. The section imageable is the name of the morphing method, and the suffix's ID and type are the keys that will define the exact ID and type of the item that it will be morphed to.

Mass assignment

When creating a new Model Instance (when inserting or updating data), we pass a variable that is set as an array with attribute names and values. These attributes are then assigned to the Model by mass assignment. If we blindly add all of the inputs into mass assignment, this will become a serious security issue. In addition to querying methods, Eloquent ORM also helps us with mass assignment. Let's say you don't want the column e-mail in your User Model (Object) to be altered in any way (blacklist), or you just want the title and body columns to be altered in your Post Model (whitelist). This can be done by setting protected $fillable and $guarded variables in your Model:

```
//User.php model
Class User Extends Eloquent {
   //Disable the mass assignment of the column email
   protected $guarded = array('email');

}


//Blog.php model
Class User Extends Eloquent {
   //Only allow title and body columns to be mass assigned
   protected $fillable = array('title', 'body');

}
```

Soft deleting

Let's say you have a `posts` table, and let's assume the data inside this table is important. Even if the `delete` command is run from the Model, you want to keep the deleted data inside your database just in case. In such cases, you can use soft deletes with Laravel.

Soft deleting doesn't actually delete the row from the table; instead, it adds a key if the data is actually deleted. When a soft deletion is made, a new column called `deleted_at` is filled with a timestamp.

To enable the soft deletes, you need to first add a timestamp column called `deleted_at` to your table (you could do this by adding `$table->softDeletes()` to your migration), then set a variable called `$softDelete` to `true` in your Model Instance.

The following is an example Model Instance for soft deletes:

```
//Post.php model
Class Post Extends Eloquent {
   //Allow Soft Deletes
   protected $softDelete = true;

}
```

Now, when you run the `delete()` method in this model, instead of actually deleting the column, it will add a `deleted_at` timestamp to it.

Now, when you run the `all()` or `get()` method, the soft-deleted columns won't be listed, like they have actually been deleted.

After such deletes, you might want to get results along with soft-deleted rows. To do this, use the `withTrashed()` method as follows:

```
$allPosts = Post::withTrashed()->get(); //These results will include
both soft-deleted and non-soft-deleted posts.
```

In some cases, you may want to fetch only soft-deleted rows. To do this, use the `onlyTrashed()` method as follows:

```
$onlySoftDeletedPosts = Post::onlyTrashed()->get();
```

To restore the soft-deleted rows, use the `restore()` method. To restore all soft-deleted posts, run a code like the following:

```
$restoreAllSoftDeletedPosts = Post::onlyTrashed()->restore();
```

To hard delete (totally delete) the soft-deleted rows from a table, use the `forceDelete()` method as follows:

```
$forceDeleteSoftDeletedPosts = Post::onlyTrashed()->forceDelete();
```

When fetching rows from a table (including soft deletes), you may want to check whether they have been soft deleted or not. This check is done by running the `trashed()` method on collection rows. This method will return a Boolean value. If true, it means the row has been soft deleted.

```
//Let's fetch a post without the soft-delete checking:
$post = Post::withTrashed()->find(1);
//Then let's check whether it's soft deleted or now
    if($post->trashed()) {
return 'This post is soft-deleted';
  } else {
   return 'This post is not soft-deleted';
}
```

Eager loading

Eloquent ORM also brings a neat solution to the N+1 query problem with **Eager Loading**. Let's assume that you have a query and loop like the following:

```
$blogs = Blog::all();
foreach($blogs as $blog) {
   var_dump($blog->images());
}
```

In this case, to access the images, one more query is executed for each loop in the backend. This will exhaust the database drastically, so to prevent this, we will use the `with()` method on the query. This will fetch all of the blogs and images, relate them in the backend, and serve them as a collection directly. Refer to the following code:

```
$blogs = Blog::with('images')->get();
foreach($blogs as $blog) {
   var_dump($blog->images);
}
```

This way, the querying will be much faster, and fewer resources will be used.

Timestamps

The main benefits of Eloquent ORM are seen when you set `$timestamps` to `true` (which is the default); you will have two columns, the first is `created_at` and the second is `updated_at`. These two columns keep the creation and last update times of data as timestamps and update them automatically on the creation or update of each row.

Query scopes

Let's say that you repeat a `where` condition several times because it's a commonly used clause in your application and this condition means something. Let's say you want to get all of the blog posts that have more than 100 views (we'll call it popular posts). Without using scopes, you'd get the posts in the following format:

```
$popularBlogPosts = Blog::where('views', '>', '100')->get();
```

However, in the example, you'll be repeating this through your application over and over again. So, why not set this as a scope? You can do this easily using Laravel's Query Scope feature.

Add the following code to your `Blog` model:

```
public function scopePopular($query) {
   return $query->where('views', '>', '100');
}
```

After doing this, you can use your scope easily with the following code:

```
$popularBlogPosts = Blog::popular()->get();
```

You can also chain the post as follows:

```
$popularBlogPosts = Blog::recent()->popular()->get();
```

Accessors and mutators

One of the features of Eloquent ORM is accessors and mutators. Let's say you have a column called `name` on your table, and on calling this column, you want to pass PHP's `ucfirst()` method to uppercase its name. This can be done by simply adding the following lines of code to the model:

```
public function getNameAttribute($value) {
    return ucfirst($value);
}
```

Now, let's consider the opposite. Each time you save or update the name column, you want to pass the PHP `strtolower()` function to the column (you want to mutate the input). This can be done by adding the following lines of code to the model:

```
public function setNameAttribute($value) {
    return strtolower($value);
}
```

Note that the method name should be CamelCased even though the column name is `snake_cased`. If your column name is `first_name`, the getter method name should be `getFirstNameAttribute`.

Model events

Model events play an important part in the Laravel design pattern. Using Model events, you can call any method right after the event is fired.

Let's say that you have set a cache for your comments, and you want to flush the cache each time a comment is deleted. How can you catch the comment's deletion event and do something there? Should there be various places in the application that such comments can be deleted? Is there a way to catch exact the "deleting" or "deleted" event? In such case, the Model events come in handy.

Models hold the following methods: `creating`, `created`, `updating`, `updated`, `saving`, `saved`, `deleting`, `deleted`, `restoring`, and `restored`.

Whenever a new item is being saved for the first time, the creating and created events will fire. If you are updating a current item on the Model, the `updating`/`updated` events will fire. Whether you are creating a new item or updating a current one, the `saving`/`saved` events will fire.

If `false` is returned from the `creating`, `updating`, `saving`, or `deleting` event, the action will be canceled.

For example, let's check whether a user has been created. If its first name is `Hybrid`, we'll cancel the creation. To add this condition, include the following lines of code in your `User` Model:

```
User::creating(function($user){
    if ($user->first_name == 'Hybrid') return false;
});
```

Model observers

Model observers are quite similar to Model Events, but the approach is a little bit different. Instead of defining all events (`creating`, `created`, `updating`, `updated`, `saving`, `saved`, `deleting`, `deleted`, `restoring`, and `restored`) inside of the Model, it "abstracts" the logic of the events to a different class and "observes" it with the `observe()` method. Let's assume we have a Model event like the following:

```
User::creating(function($user){
    if ($user->first_name == 'Hybrid') return false;
});
```

To keep the abstraction, it will be much better to wrap all of these events and separate their logic from the Model. In an observer, these events will look like the following:

```
class UserObserver {

   public function creating($model){
       if ($model->first_name == 'Hybrid') return false;
    }

    public function saving($model)
    {
        //Another model event action here
    }

}
```

As you can imagine, you can put this class anywhere in your application. You can even group all of these events in a separate folder for a better architectural pattern.

Now, you need to register this event `Observer` class to a Model. This can be done with the following simple command:

```
User::observe(new UserObserver);
```

The main advantage of this approach is that you can use observers in more than one Model and register more than one observer to a Model this way.

Migrations

Migrations are easy tools to version control your database. Let's say there is a place where you need to add a new column to the table or roll back to the previous state because you did something wrong or the link to your application broke. Without migrations, these are tedious tasks to handle, but with migrations, your life will be much easier.

There are various reasons to use migrations; some of these are as follows:

- You'll benefit from this versioning system. If you made a mistake or need to roll back to a previous state, you can do so with only a single command using migrations.
- The use of migrations for alteration will bring about flexibility. The migrations that are written will work on all supported database drivers, so you won't need to rewrite database code again and again for different drivers. Laravel will handle this in the background.

- They are quite easy to generate. Using the migration commands of the Laravel `php` client, which is called `artisan`, you can manage all of your application's migrations.

The following is what a migration file looks like:

```
<?php

use Illuminate\Database\Migrations\Migration;

class CreateNewsTable extends Migration {

        /**
        * Run the migrations.
        */
        public function up()
        {
                //
        }

        /**
        * Reverse the migrations.
        */
        public function down()
        {
                //
        }

}
```

The `up()` method runs when the migration is run forward (a new migration). The `down()` method runs when the migration is run backward, meaning it reverses or resets (reverses and reruns) the migration.

After these methods are triggered via the `artisan` command, it runs the method up or down, corresponding the parameters of the `artisan` command, and returns the status of the message.

Database seeders

Let's say you've programmed a blog application. You need to show what it's capable of, but there are no example blog posts to show the awesome blog you've programmed. This is where seeders come in handy.

Database seeders are some simple classes that fill random data in a specified table. The seeder class has a simple method called `run()` to make this seeding(s). The following is what a seeder looks like:

```
<?php

class BlogTableSeeder extends Seeder {

  public function run()
  {
    DB::table('blogs')->insert(array(
      array('title' => 'My Title'),
      array('title' => 'My Second Title'),
      array('title' => 'My Third Title')
    ));
  }

}
```

When you call this class from a terminal using the `artisan` command, it connects to the database and fills it with the given data. After this attempt, it returns a command message to the user over the terminal about the status of the seeding.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Summary

In this chapter, we have learned about the role of the Model in the MVC pattern and how Laravel 4 "defines" the Model by extending its roles to various classes. We've also seen what the Model components of Laravel 4 are capable of with examples.

In the next chapter, we'll learn about the role of the View and how it interacts with end users and other aspects of the application using the MVC pattern on Laravel 4.