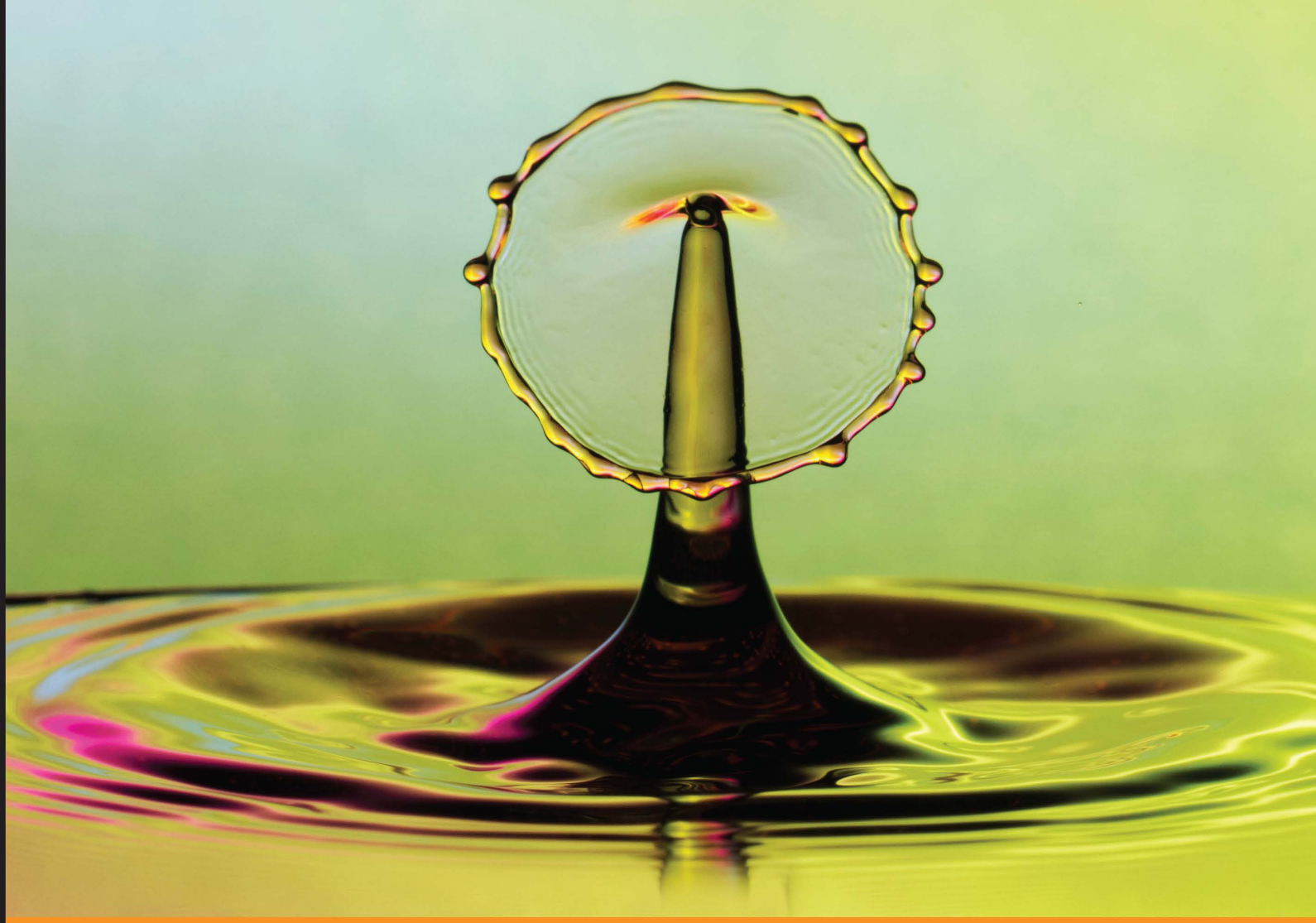


Community Experience Distilled

Laravel Application Development Blueprints

Learn to develop 10 fantastic applications with the new and improved Laravel 4

Arda Kılıçdağı
Halil İbrahim Yılmaz

[PACKT] PUBLISHING open source* community experience distilled

Laravel Application Development Blueprints

Learn to develop 10 fantastic applications with the new and improved Laravel 4

Arda Kılıçdağı

Halil İbrahim Yılmaz

[PACKT] open source
community experience distilled
PUBLISHING

BIRMINGHAM - MUMBAI

Laravel Application Development Blueprints

First published: November 2013

Production Reference: 1071113

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78328-211-1

www.packtpub.com

Cover Image by Jarek Blaminsky (milak6@wp.pl)

Credits

Authors

Arda Kılıçdağı

Halil İbrahim Yılmaz

Reviewers

James Blackwell

Mhd Zaher Ghaibeh

Acquisition Editor

Kevin Colaco

Lead Technical Editor

Ankita Shashi

Technical Editors

Vrinda Nitesh Bhosale

Nikhita K. Gaikwad

Rahul U. Nair

Copy Editors

Alisha Aranha

Janbal Dharmaraj

Tanvi Gaitonde

Gladson Monteiro

Alfida Paiva

Adithi Shetty

Project Coordinator

Wendell Palmer

Proofreader

Joanna McMahon

Indexer

Hemangini Bari

Production Coordinator

Aditi Gajjar

Cover Work

Aditi Gajjar

About the Authors

Arda Kılıçdağı is a PHP, MySQL, and JavaScript programmer from Turkey. He has been developing applications in PHP since 2005. He has been administrating the Turkish national support website for the well-known open source content management script, PHP-Fusion. He's also one of the international developers and a member of the management team for PHP-Fusion, and he has an important role in the project'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 is experienced in using the Facebook API, Twitter API, and PayPal's Adaptive Payments API (which is used on crowdfunding websites such as KickStarter). He's also experienced in using JavaScript, and he's currently infusing his applications with JavaScript and jQuery, both on the frontend and backend.

He has also developed applications using CodeIgniter and CakePHP for about four years, but these PHP frameworks didn't suit his needs completely. This is why he decided to use another framework for his projects, and that is when he met Laravel. Currently he is developing all his applications using Laravel.

He's also obsessed with Unix/Linux and uses Linux on a daily basis. In addition, he is administrating the world's best-known microcomputer, Raspberry Pi's biggest Turkish community website.

I'd like to thank to my mother and father, Serhan Karakaya, Barkev Keskin, Alpbuğra Bahadır Gültekin, Ferdi, Mrs. Deger Dundar, Mr. Orkun Altinbayrak, and all my other friends who I cannot list, for their support and understanding.

Halil İbrahim Yılmaz is a Python and PHP programmer and an e-commerce consultant from Turkey. He has worked as a developer and a software coordinator in over a dozen ventures, including Begendy, Modeum, Futbolkurdu, Arkeoidea, and Uzmanlazim. He is experienced in using many APIs such as Google, YouTube, Facebook, Twitter, Grooveshark, and PayPal. After meeting his business partner, he co-founded 16 Pixel, a Bursa-based creative consultancy that specializes in web development and e-commerce.

He loves learning functional programming languages (Erlang and Haskell), new JavaScript technologies (Node.js), and NoSQL database systems (Riak and MongoDB). When he is not working on client projects, he is often trying to code a web application with those technologies.

He lives in a house full of Linux boxes in Bursa, Turkey.

I'd like to thank my daughter İklim for her presence, and Gezi Park protestors for their cause to make the world a better place.

About the Reviewers

James Blackwell is a full stack, freelance web developer with years of experience in producing web applications. He's produced and worked on many large websites and applications for a range of companies with multiple technologies such as PHP, JavaScript, MySQL, and MongoDB.

Mhd Zaher Ghaibeh is the co-founder of Creative Web Group, Syria (`http://creativewebgroup-sy.com/`), a web development startup that specializes in developing modern web applications and utilizes the latest web development technologies and methodologies. He has over eight years of web development experience and holds a Bachelor of Information Technology degree from Syrian University, Damascus.

He is currently working with Tipsy & Tumbler Limited (`http://www.tipsyandtumbler.co.uk/`) as a PHP web developer.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Laravel Application Development Blueprints covers how to develop 10 different applications step-by-step using Laravel 4. You will also learn about both basic and advanced usage of Laravel's built-in methods, which will come in handy for your project. Also, you will learn how to extend the current libraries with the built-in methods and include third-party libraries.

This book looks at the Laravel PHP framework and breaks down the ingrained prejudice that coding with PHP causes spaghetti code. It will take you through a number of clear, practical applications that will help you take advantage of the Laravel PHP framework and PHP OOP programming, while avoiding spaghetti code.

You'll also learn about creating secure web applications using different methods, such as file uploading and processing, making RESTful Ajax requests, and form processing. If you want to take advantage of the Laravel PHP framework's validate, file processing, and RESTful controllers in various types of projects, this is the book for you. Everything you need to know to code fast and secure applications with the Laravel PHP framework will be discussed in this book.

What this book covers

Chapter 1, *Building a URL Shortener Website*, provides an overview of the very basics of Laravel 4. This chapter introduces the basics of routes, migrations, models, and views.

Chapter 2, *Building a To-do List with Ajax*, uses the Laravel PHP framework and jQuery to build the application. Through out this chapter, we'll show you the basics of RESTful controllers, RESTful routing, and validating request types.

Chapter 3, *Building an Image Sharing Website*, covers how to add a third-party library to the project, and how to upload, resize, process, and show images.

Chapter 4, *Building a Personal Blog*, covers how to code a simple personal blog with Laravel. Throughout the chapter, Laravel's built-in authentication, paginate mechanism, and the named routes' features are covered. In this chapter, we'll also elaborate on some rapid development methods that come with Laravel, such as methods to easily create a URL for routes.

Chapter 5, *Building a News Aggregation Website*, focuses mainly on extending the core classes with custom functions and using them. The usage of migrations and the basics of validating, saving, and retrieving data is also covered.

Chapter 6, *Creating a Photo Gallery System*, helps us code a simple photo gallery system with Laravel. In this chapter, we'll cover Laravel's built-in file validation, file upload, and the **hasMany** database relation method. We'll also cover the validation class for validating the data and the uploaded files. Finally, we'll elaborate on Laravel's file class for processing the files.

Chapter 7, *Creating a Newsletter System*, covers an advanced newsletter system, which will use Laravel's queue and e-mail libraries. This chapter also focuses on how to set and fire/trigger queued tasks, and how to parse e-mail templates and send mass e-mails to subscribers.

Chapter 8, *Building a Q&A Web Application*, mainly focuses on pivot tables, why and where they are needed, and their usage. This chapter also covers the usage of a third-party authentication system and methods to remove or rename the public segment.

Chapter 9, *Building a RESTful API – The Movies and Actors Database*, focuses on the basics of REST by coding a simple movies and actors API with Laravel. We'll make some JSON endpoints behind a basic authentication system and learn a few Laravel 4 tricks throughout the chapter. Also, we'll cover the basics of RESTful controllers, RESTful routing, and adding sample data to the database with migrations.

Chapter 10, *Building an E-commerce Website*, discusses how to code a simple e-commerce application with Laravel. In this chapter we'll cover Laravel's built-in, basic-authentication mechanism named routes and database seeding. We will also elaborate on some rapid development methods that come with Laravel 4. We'll also cover advanced usage of pivot tables. Our e-commerce application will be a simple bookstore. The application will have order, administration, and cart features.

What you need for this book

The applications written in the chapters are all based on Laravel 4, so you will require what's listed on Laravel 4's standard requirements list, which will be available at `http://four.laravel.com/docs#server-requirements`.

The chapter requirements are as follows:

- PHP 5.3.7 or above
- MCrypt PHP Extension
- A SQL database to store the data

There may be additional requirements for individual third-party packages. Please refer to their requirements page if they are used anywhere during the chapters.

Who this book is for

This book is great for developers new to the PHP 5 object-oriented programming standards and who are looking to work with the Laravel PHP framework. It's assumed that you will have some experience in PHP already, as well as being familiar with coding current "old school" methods, such as not using any PHP framework. This book is also for those who are already using a PHP framework and are looking for a better solution.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
<?php
class Todo extends Eloquent
{
  protected $table = 'todos';

}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
  public function run()
  {
    Eloquent::unguard();
    $this->call('UsersTableSeeder');
    $this->command->info('Users table seeded!');
```

```
        $this->call('AuthorsTableSeeder');
        $this->command->info('Authors table seeded!');
    }

}
```

Any command-line input or output is written as follows:

```
php artisan migrate
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Then, we check whether the user who has clicked on the **best answer** button is either the poser of the question or the application's administrator."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

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.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Building a URL Shortener Website

Throughout the book, we will be using the Laravel PHP framework to build different types of web projects.

In this chapter, we'll see how to build a URL Shortener website with the basics of Laravel framework. The covered topics include:

- Creating a database and migrating our URL Shortener's table
- Creating our form
- Creating our Link model
- Saving data to the database
- Getting individual URL from the database and redirecting

Creating a database and migrating our URL shortener's table

Migrations are like version control for your application's database. They permit a team (or yourself) to modify the database schema, and provide up-to-date information on the current schema state. To create and migrate your URL Shortener's database, perform the following steps:

1. First of all, we have to create a database, and define the connection information to Laravel. To do this, we open the `database.php` file under `app/config`, and then fill the required credentials. Laravel supports MySQL, SQLite, PostgreSQL, and SQLSRV (Microsoft SQL Server) by default. For this tutorial, we will be using MySQL.

2. We will have to create a MySQL database. To do this, open your MySQL console (or phpMyAdmin), and write down the following query:

```
CREATE DATABASE urls
```

3. The previous command will generate a new MySQL database named `urls` for us. After having successfully generated the database, we will be defining the database credentials. To do this, open your `database.php` file under `app/config`. In that file, you will see multiple arrays being returned with database definitions.
4. The `default` key defines what database driver to be used, and each database driver key holds individual credentials. We just need to fill the one that we will be using. In our case, I've made sure that the default key's value is `mysql`. To set the connection credentials, we will be filling the value of the `mysql` key with our database name, username, and password. In our case, since we have a `database` named `urls`, with the `username` as `root` and without a password, our `mysql` connection settings in the `database.php` file will be as follows:

```
'mysql' => array(
  'driver' => 'mysql',
  'host' => 'localhost',
  'database' => 'database',
  'username' => 'root',
  'password' => '',
  'charset' => 'utf8',
  'collation' => 'utf8_unicode_ci',
  'prefix' => '',
),
```

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.

5. Now, we will be using the **Artisan CLI** to create migrations. Artisan is a command-line interface specially made for Laravel. It provides numerous helpful commands to help us in development. We'll be using the following `migrate:make` command to create a migration on Artisan:

```
php artisan migrate:make create_links_table --table=
  links --create
```

The command has two parts:

- The first is `migrate:make create_links_table`. This part of the command creates a migration file which is named something like `2013_05_01_194506_create_links_table.php`. We'll examine the file further.
- The second part of the command is `--table=links --create`.
  - The `--table=links` option points to the database name.
  - The `--create` option is for creating the table on the database server to which we've given the `--table=links` option.

6. As you can see, unlike Laravel 3, when you run the previous command, it will create both the migrations table and our migration. You can access the migration file under `app/database/migrations`, having the following code:

```
<?php
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;
class CreateLinksTable extends Migration {
  /**
  * Run the migrations.
  *
  * @return void
  */
  public function up()
  {
    Schema::create('links', function(Blueprint $table)
    {
      $table->increments('id');
    });
  }
  /**
  * Reverse the migrations.
  *
  * @return void
  */
  public function down()
  {
    Schema::drop('links');
  }
}
```

7. Let's inspect the sample migration file. There are two public functions which are declared as `up()` and `down()`. When you execute the following `migrate` command, the contents of the `up()` function will be executed:

   ```
   php artsian migrate
   ```

   This command will execute all of the migrations and create the `links` table in our case.

If you receive a `class not found` error when running the migration file, try running the `composer update` command.

8. We can also roll back to the last migration, like it was never executed in the first place. We can do this by using the following command:

   ```
   php artisan migrate:rollback
   ```

9. In some cases, we may also want to roll back all migrations we have created. This can be done with the following command:

   ```
   php artisan migrate:reset
   ```

10. While in the development stages, we may forget to add/remove some fields, or even forget to create some tables, and we may want to roll back everything and remigrate them all. This can be done using the following command:

    ```
    php artisan migrate:refresh
    ```

11. Now, let's add our fields. We've created two additional fields called `url` and `hash`. The `url` field will hold the actual URL, to which the URL present in the `hash` field will be redirected. The `hash` field will hold the shortened version of the URL that is present in the `url` field. The final content of the migration file is as shown in the following code:

    ```
    <?php
    use Illuminate\Database\Migrations\Migration;
    class CreateLinksTable extends Migration {
      /**
      * Run the migrations.
      *
      * @return void
      */
      public function up()
      {
        Schema::create('links', function(Blueprint $table)
        {
    ```

```
            $table->increments('id');
            $table->text('url');
            $table->string('hash',400);
        });
    }
    /**
    * Reverse the migrations.
    *
    * @return void
    */
    public function down()
    {
        Schema::drop('links');
    }
}
```

Creating our form

Now let's make our first form view.

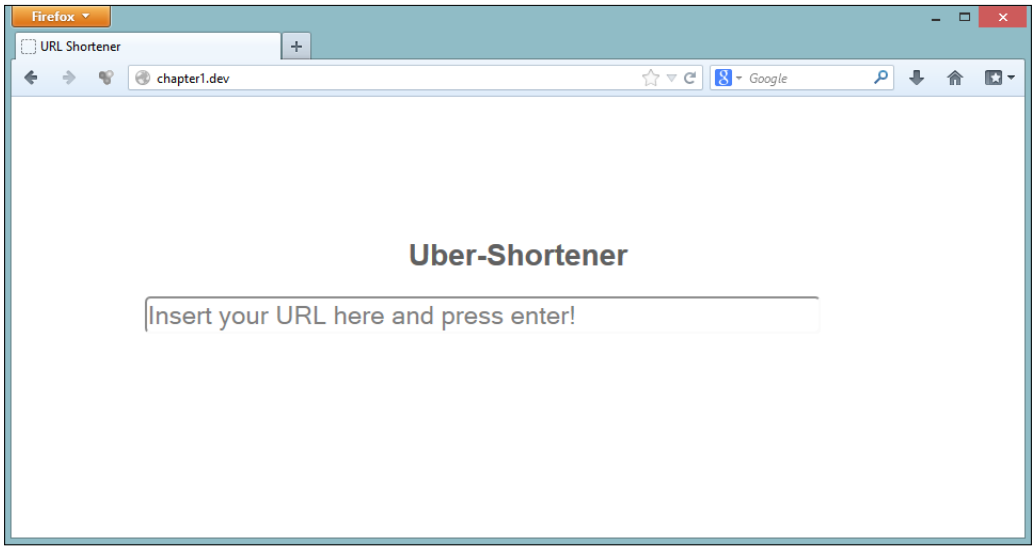
1. Save the following code as `form.blade.php` under `app/views`. The file's extension is `blade.php` because we will be benefiting from Laravel 4's built-in template engine called **Blade**. There may be some methods you don't understand in the form yet, but don't worry. We will cover everything regarding this form in this chapter.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>URL Shortener</title>
    <link rel="stylesheet" href="/assets/css/styles.css" />
  </head>
  <body>
    <div id="container">
      <h2>Uber-Shortener</h2>
      {{Form::open(array('url'=>'/','method'=>'post'))}}

      {{Form::text('link',Input::old('link'),
        array('placeholder'=>
        'Insert your URL here and press enter!'))}}
      {{Form::close()}}
    </div>
  </body>
</html>
```

2. Now save the following codes as `styles.css` under `public/assets/css`:

```
div#container{padding-top:100px;
  text-align:center;
  width:75%;
  margin:auto;
  border-radius:4px}
div#container h2{font-family:Arial,sans-serif;
  font-size:28px;
  color:#555}
div#container h3{font-family:Arial,sans-serif;
  font-size:28px}
div#container h3.error{color:#a00}
div#container h3.success{color:#0a0}
div#container input{display:block;
  width:90%;
  float:left;
  font-size:24px;
  border-radius:5px}
div#error,div#success{border-radius:3px;
  display:block;
  width:90%;
  padding:10px}
div#error{background:#ff8080;
  border:1px solid red}
div#success{background:#80ff80;
  border:1px solid #0f0}
```

This code will produce you a form that looks like the following screenshot:



As you can see, we have used a CSS file to tidy up the form a bit, but the actual part of the form is at the bottom of the `View` file, inside `div` with the ID of the container.

3. We have used the Laravel's built-in `Form` class to generate a form, and we have used the `old()` method of the `Input` library. Now let's dig the code:
    - `Form::open()`: It creates a `<form>` opening tag. Within the first provided parameter, you can define how the form is sent, and where it is going to be sent. It can be a controller's action, a direct URL, or a named route.
    - `Form::text()`: It creates an `<input>` tag with type as text. The first parameter is the name of the input, the second parameter is the value of the input, and within the array given in the third parameter, you can define assets and other attributes of the `<input>` tag.
    - `Input::old()`: It will return the old input from a form, after the form is returned with the inputs. The first parameter is the name of the old input submitted. In our case, if the form is returned after submission (for example, if the form validation fails), the text field will be filled with our old input and we can reuse it for later requests.
    - `Form::close()`: It closes the `<form>` tag.

Creating our Link model

To benefit from Laravel's ORM class called `Eloquent`, we need to define a model. Save the following code as `Link.php` under `app/models`:

```
<?php
class Link extends Eloquent {
  protected $table = 'links';
  protected $fillable = array('url','hash');
  public $timestamps = false;
}
```

The Eloquent model is quite easy to understand.

- The variable `$table` is used to define the table name for the model, but it's not compulsory. Even if we don't define this variable, it would use the plural form of the model name as a database table name. For example, if the model name was post, it would look for the post's table as default. This way, you can use any model names for the tables.

- The protected `$fillable` variable defines what columns can be (mass) created and updated. Laravel 4 blocks the mass-assignment of the values of all the columns with `Eloquent` by default. This way, for example, if you have a `users` table, and you are the only user/administrator, the mass-assignment protects your database from another user being added.
- The public `$timestamps` variable checks whether the model should try setting the timestamps `created_at` and `updated_at` by default, while creating and updating the query respectively. Since we don't need these features for our chapter, we will disable this by setting the value to `false`.

We now need to define this view to show whether we can navigate to our virtual host's index page. You can do this either from the controllers defined in `routes.php`, or from `routes.php` directly. Since our application is small, defining them directly from `routes.php` should suffice. To define this, open the `routes.php` file under the `app` folder and add the following code:

```
Route::get('/', function()
{
  return View::make('form');
});
```

If you already have a section starting with `Route::get('/', function()`, you should replace that section with the previous code.

Laravel can listen `get`, `post`, `put`, and `delete` requests. Since our action is a `get` action (because we will be navigating through the browser without posting), our route type will be `get`, and because we want to show the view on the root page, our first parameter of the `Route::get()` method will be `/`, and we wrap a closure function as the second parameter to define what we want to do. In our case, we will be returning `form.blade.php` placed under `app/views` that we had generated before, so we just type `return View::make('form')`. This method returns the `form.blade.php` view from the `views` folder.

If the view was in a subdirectory, it would be called `subfolder.form`.

Saving data to the database

Now we need to write a route that will have to listen to our `post` request. For this, we open our `routes.php` file under the `app` folder and add the following code:

```
Route::post('/',function(){
  //We first define the Form validation rule(s)
  $rules = array(
    'link' => 'required|url'
  );
  //Then we run the form validation
  $validation = Validator::make(Input::all(),$rules);
  //If validation fails, we return to the main page with an
    error info
  if($validation->fails()) {
    return Redirect::to('/')
    ->withInput()
    ->withErrors($validation);
  } else {
    //Now let's check if we already have the link in
      our database. If so, we get the first result
    $link = Link::where('url','=',Input::get('link'))
    ->first();
    //If we have the URL saved in our database already, we
      provide that information back to view.
    if($link) {
      return Redirect::to('/')
      ->withInput()
      ->with('link',$link->hash);
      //Else we create a new unique URL
    } else {
      //First we create a new unique Hash
      do {
        $newHash = Str::random(6);
      } while(Link::where('hash','=',$newHash)
        ->count() > 0);

      //Now we create a new database record
      Link::create(array(
        'url' => Input::get('link'),
        'hash' => $newHash
      ));
```

```
        //And then we return the new shortened URL info to
          our action
        return Redirect::to('/')
        ->withInput()
        ->with('link',$newHash);
      }
    }
});
```

Validating the users' input

Using the `post` action function that we've coded now, we will be validating the user's input with the Laravel's built-in `Validation` class. This class helps us prevent invalid inputs from getting into our database.

We first define a `$rules` array to set the rules for each field. In our case, we want the link to have a valid URL format. Then we can run the form validation using the `Validator::make()` method and assign it to the `$validation` variable. Let's understand the parameters of the `Validator::make()` method:

- The first parameter of the `Validator::make()` method takes an array of inputs and values to be validated. In our case, the whole form has only one field called link, so we've put the `Input::all()` method, which returns all the inputs from the form.
- The second parameter takes the validation rules to be checked. The stored `$validation` variable holds some information for us. For example, we can check whether the validation has failed or passed (using `$validation->fails()` and `$validation->passes()`). These two methods return Boolean results, so we can easily check if the validation has passed or failed. Also, the `$validation` variable holds a method `messages()`, which contains the information of a failed validation. And we can catch them using `$validation->messages()`.

If the form validation fails, we redirect the user back to our index page (`return Redirect::to('/')`), which holds the URL shortener form, and we return some flash data back to the form. In Laravel, we do this by adding the `withVariableName` object to the redirected page. Using `with` is mandatory here, which will tell Laravel that we are returning something additional. We can do this for both redirecting and making views. If we are making views and showing some content to the end user, those `withVariableName` will be variables, and we can call them directly using `$VariableName`, but if we are redirecting to a page with the `withVariableName` object, `VariableName` will be a flash session data, and we can call it using the `Session` class (`Session::get('VariableName')`).

In our example, to return the errors, we used a special method `withErrors($validation)`, instead of returning `$validation->messages()`. We could also return using that, but the `$errors` variable is always defined on views, so we can use our `$validation` variable as a parameter with `withErrors()` directly. The `withInput()` method is also a special method, which will return the results back to the form.

```
//If validation fails, we return to the main page with
  an error info
if($validation->fails()) {
  return Redirect::to('/')
  ->withInput()
  ->withErrors($validation);
}
```

If the user forgets one field in the form, and if the validation fails and shows the form again with error messages, using the `withInput()` method, the form can be filled with the old inputs again. To show these old inputs in Laravel, we use the `old()` method of the `Input` class. For example, `Input::old('link')` will return us the old input of the form field named `link`.

Returning the messages to the view

To return the error message back to the form, we can add the following HTML code into `form.blade.php`:

```
@if(Session::has('errors'))
<h3 class="error">{{$errors->first('link')}}</h3>
@endif
```

As you can already guess, `Session::has('variableName')` returns a Boolean value to check whether there is a variable name for the session. Then, with the `first('formFieldName')` method of Laravel's `Validator` class, we are returning the first error message of a form field. In our case, we are showing the first error message of our `link` form field.

Diving further into controller and processing the form

The `else` part of the validation checking part that is executed when the form validation is completed successfully in our example, holds the further processing of the link. In this section, we will perform the following steps:

1. Checking whether the link is already in our database.
2. If the link is already in our database, returning the shortened link.
3. If the link is not present in our database, creating a new random string (the hash that will be in our URL) for the link.
4. Creating a new record in our database with the provided values.
5. Returning the shortened link back to the user.

Now, let's dig the code.

1. Here's the first part of our code:

```
// Now let's check if we already have the link in our
  database. If so, we get the first result
$link = Link::where('url','=',Input::get('link'))
->first();
```

   First, we check if the URL is already present in our database using the `where()` method of **Fluent Query Builder**, and get the first result via the method `first()`, and assign it to the `$link` variable. You can use the Fluent query methods along with the Eloquent ORM easily. If this confuses you, don't worry, we will cover this further in the later chapters.

2. This is the next part of our controller method's code:

```
//If we have the URL saved in our database already, we
  provide that information back to view.
if($link) {
  return Redirect::to('/')
  ->withInput()
  ->with('link',$link->hash);
```

   If we have the URL saved in our database, the `$link` variable will hold the object of our link's information taken from the database. So with a simple `if()` clause, we can check if there is a result. If there is a result returned, we can access it using `$link->columnname`.

In our case, if the query has a result, we redirect the inputs and the link back to the form. As we've used here, the `with()` method can also be used with two parameters instead of using a camel case—`withName('value')` is exactly the same as `with('name','value')`. So, we can return the hash code with a flash data named link `with('link',$link->hash)`. To show this, we can add the following code to our form:

```
@if(Session::has('link'))
<h3 class="success">
  {{Html::link(Session::get('link'),'Click here for your
    shortened URL')}}
</h3>
@endif
```

The `Html` class helps us write HTML codes easily. The `link()` method requires the following two parameters:

- The first parameter is `link`. If we provide a string directly (the hash string in our case), the class will identify it automatically and make an internal URL from our website.
- The second parameter is the string that has the link.

The optional third parameter has to be an array, holding attributes (such as class, ID, and target) as a two-dimensional array.

3. The following is the next part of our code:

```
//Else we create a new unique URL
} else {
  //First we create a new unique Hash
  do {
    $newHash = Str::random(6);
  } while(Link::where('hash','=',$newHash)->count() > 0);
```

If there is no result (the else clause of the variable), we are creating a six-character-long alphanumeric random string with the `Str` class's `random()` method and checking it each time to make sure that it is a unique string, using PHP's own do-while statement. For real world application, you can use an alternative method to shorten, for example, converting an entry in the ID column to base_62 and using it as a hash value. This way, the URL would be cleaner, and it's always a better practice.

4. This is the next part of our code:

```
//Now we create a new database record
Link::create(array(
  'url' => Input::get('link'),
  'hash' => $newHash
));
```

Once we have a unique hash, we can add the link and the hash values to the database with the `create()` method of the Laravel's Eloquent ORM. The only parameter should be a two-dimensional array, in which the keys of the array are holding the database column names, and the values of the array are holding the values to be inserted as a new row.

In our case, the `url` column has to have the `link` field's value that came from the form. We can catch these values that came from the `post` request using Laravel's `Input` class's `get()` method. In our case, we can catch the value of the `link` form field that came from the `post` request (which we would catch using the spaghetti code `$_POST['link']`) using `Input::get('link')`, and return the hash value to the view as we did earlier.

5. This is the final part of our code:

```
//And then we return the new shortened URL info to our
  action return Redirect::to('/')
->withInput()
->with('link',$newHash);
```

Now, at the output, we're redirected to `oursite.dev/hashcode`. There is a link stored in the variable `$newHash`; we need to catch this hash code and query our database, and if there is a record, we need to redirect to the actual URL.

Getting individual URL from the database and redirecting

Now, in the final part of our first chapter, we need to get the `hash` part from the generated URL, and if there is a value, we need to redirect it to the URL which is stored in our database. To do this, add the following code at the end of your `routes.php` file under the `app` folder:

```
Route::get('{hash}',function($hash) {
  //First we check if the hash is from a URL from our
    database
  $link = Link::where('hash','=',$hash)
  ->first();
  //If found, we redirect to the URL
  if($link) {
    return Redirect::to($link->url);
    //If not found, we redirect to index page with error
      message
  } else {
```

```
        return Redirect::to('/')
        ->with('message','Invalid Link');
      }
  })->where('hash', '[0-9a-zA-Z]{6}');
```

In the previous code, unlike other route definitions, we added curly brackets around the name `hash`, which tells Laravel that it's a parameter; and with the `where()` method we defined how the name parameter will be. The first parameter is the name of the variable (which is `hash` in our case), and the second parameter is a regular expression that will filter the parameter. In our case, the regular expression filters an exact alphanumeric string that is six-characters long. This way, we can filter our URLs and secure them from start, and we won't have to check if the `url` parameter has something we don't want (for example, if alphabets are entered instead of numbers in the ID column). To get individual URL from the database and redirect, we perform the following steps:

1. In the `Route` class, we first make a search query, as we did in the earlier section, and check if we have a link with the given hash from a URL in our database, and set it to a variable called `$link`.

   ```
   //First we check if the hash is from an URL from our
     database
   $link = Link::where('hash','=',$hash)
   ->first();
   ```

2. If there is a result, we redirect the page to the `url` column of our database, which has the link to which the user should be redirected.

   ```
   //If found, we redirect to the link
   if($link) {
     return Redirect::to($link->url);
   }
   ```

3. If there is no result, we redirect the user back to our index page using the `$message` variable, which holds the value `Invalid Link`.

   ```
   //If not found, we redirect to index page with error message
   } else {
     return Redirect::to('/')
     ->with('message','Invalid Link');
   }
   ```

   To show the `Invalid Link` message in the form, add the following code in your `form.blade.php` file placed under `app/views`:

   ```
   @if(Session::has('message'))
   <h3 class="error">{{Session::get('message')}}</h3>
   @endif
   ```

Summary

In this chapter, we have covered the basic usage of Laravel's routes, models, artisan commands, and database drivers by making a simple URL shortener website. Once you've followed this chapter, you can now create database tables with migrations, write simple forms with the Laravel Form Builder Class, validate these forms with the `Validation` class, and process these forms and insert new data to the table(s) with the Fluent Query Builder or Eloquent ORM. In the next chapter, we'll cover the advanced usage of these awesome features.

2
Building a To-do List with Ajax

In this chapter, we will be using the Laravel PHP framework and jQuery to build a to-do list with Ajax.

Through out this chapter, we'll show you the basics of **RESTful controllers**, **RESTful routing**, and **Request types**. The list of topics covered in this chapter is as follows:

- Creating and migrating our to-do list's database
- Creating a to-do list's model
- Creating the template
- Inserting data to the database with Ajax
- Retrieving the list from the database
- How to allow only Ajax requests

Creating and migrating our to-do list's database

As you know from the previous chapter, migrations are very helpful to control development steps. We'll use migrations again in this chapter.

To create our first migration, type the following command:

```
php artisan migrate:make create_todos_table --table=todos --create
```

When you run this command, **Artisan** will generate a migration to generate a database table named `todos`.

Now we should edit the migration file for the necessary database table columns. When you open the folder `migration` in `app/database/` with a file manager, you will see the migration file under it.

Let's open and edit the file as follows:

```
<?php
use Illuminate\Database\Migrations\Migration;
class CreateTodosTable extends Migration {

    /**
    * Run the migrations.
    *
    * @return void
    */
    public function up()
    {

        Schema::create('todos', function(Blueprint $table){
            $table->create();
            $table->increments("id");
            $table->string("title", 255);
            $table->enum('status', array('0', '1'))->default('0');
            $table->timestamps();
        });

    }

    /**
    * Reverse the migrations.
    *
    * @return void
    */
    public function down()
    {
        Schema::drop("todos");
    }

}
```

To build a simple to-do list, we need five columns:

- The `id` column will store ID numbers of to-do tasks
- The `title` column will store a to-do task's title

- The `status` column will store the status of each task
- The `created_at` and `updated_at` columns will store the created and updated dates of tasks

If you write `$table->timestamps()` in the migration file, Laravel's `migration`

class automatically creates `created_at` and `updated_at` columns. As you know from *Chapter 1*, *Building a URL Shortener Website*, to apply migrations, we should run the following command:

```
php artisan migrate
```

After the command is run, if you check your database, you will see that our `todos` table and columns have been created. Now we need to write our model.

Creating a todos model

To create a model, you should open the `app/models/` directory with your file manager. Create a file named `Todo.php` under the directory and write the following code:

```
<?php
class Todo extends Eloquent
{
  protected $table = 'todos';

}
```

Let's examine the `Todo.php` file.

As you see, our `Todo` class extends an `Eloquent` model, which is the **ORM** (**Object Relational Mapper**) database class of Laravel.

The `protected $table = 'todos';` code tells `Eloquent` about our model's table name. If we don't set the `table` variable, `Eloquent` accepts the plural version of the lower case model name as the table name. So this isn't required technically.

Now, our application needs a template file, so let's create it.

Creating the template

Laravel uses a template engine that is called **Blade** for static and application template files. Laravel calls the template files from the `app/views/` directory, so we need to create our first template under this directory.

1. Create a file with the name `index.blade.php`.
2. The file contains the following code:

```
<html>
  <head>
    <title>To-do List Application</title>
    <link rel="stylesheet" href="assets/css/style.css">
    <!--[if lt IE 9]><script
      src="//html5shim.googlecode.com/svn/trunk/html5.js">
      </script><![endif]-->

  </head>
  <body>
    <div class="container">
      <section id="data_section" class="todo">
        <ul class="todo-controls">
        <li><img src="/assets/img/add.png" width="14px"
        onClick="show_form('add_task');" /></li>
        </ul>
          <ul id="task_list" class="todo-list">
          @foreach($todos as $todo)
            @if($todo->status)
              <li id="{{$todo->id}}" class="done">
                <a href="#" class="toggle"></a>
                <span id="span_{{$todo->id}}">{
                {$todo->title}}</span> <a href="#"
                onClick="delete_task('{{$todo->id}}');"
                class="icon-delete">Delete</a> <a href="#"
                onClick="edit_task('{{$todo->id}}',
                '{{$todo->title}}');"
                class="icon-edit">Edit</a></li>
            @else
              <li id="{{$todo->id}}"><a href="#"
                onClick="task_done('{{$todo->id}}');"
                class="toggle"></a> <span id="span_{
                {$todo->id}}">{{$todo->title}}</span>
                <a href="#" onClick="delete_task('{
                {$todo->id}}');" class=
                "icon-delete">Delete</a>
```

```
                    <a href="#" onClick="edit_task('{
                    {$todo->id}}','{{$todo->title}}');"
                    class="icon-edit">Edit</a></li>
                @endif
              @endforeach
            </ul>
          </section>
          <section id="form_section">

          <form id="add_task" class="todo"
            style="display:none">
          <input id="task_title" type="text" name="title"
            placeholder="Enter a task name" value=""/>
          <button name="submit">Add Task</button>
          </form>

            <form id="edit_task" class="todo"
              style="display:none">
            <input id="edit_task_id" type="hidden" value="" />
            <input id="edit_task_title" type="text"
              name="title" value="" />
            <button name="submit">Edit Task</button>
          </form>

        </section>

        </div>
        <script src="http://code.jquery.com/
          jquery-latest.min.js"type="text/javascript"></script>
        <script src="assets/js/todo.js"
          type="text/javascript"></script>
      </body>
    </html>
```

The preceding code may be difficult to understand if you're writing a Blade template for the first time, so we'll try to examine it. You see a `foreach` loop in the file. This statement loops our `todo` records.

We will provide you with more knowledge about it when we are creating our controller in this chapter.

`If` and `else` statements are used for separating finished and waiting tasks. We use `if` and `else` statements for styling the tasks.

We need one more template file for appending new records to the task list on the fly. Create a file with the name `ajaxData.blade.php` under the `app/views/` folder. The file contains the following code:

```
@foreach($todos as $todo)
  <li id="{{$todo->id}}"><a href="#" onClick="task_done('{{$todo-
  >id}}');" class="toggle"></a> <span id="span_{{$todo
  >id}}">{{$todo->title}}</span> <a href="#"
  onClick="delete_task('{{$todo->id}}');" class="icon
  delete">Delete</a> <a href="#" onClick="edit_task('{{$todo
  >id}}','{{$todo->title}}');" class="icon-edit">Edit</a></li>
@endforeach
```

Also, you will see the `/assets/` directory in the source path of static files. When you look at the `app/views` directory, there is no directory named `assets`. Laravel separates the system and public files. Public accessible files stay under your `public` folder in `root`. So you should create a directory under your public folder for `asset` files.

We recommend working with these types of organized folders for developing tidy and easy-to-read code. Finally, you will see that we are calling jQuery from its main website. We also recommend this way for getting the latest, stable jQuery in your application.

You can style your application as you wish, hence we'll not examine styling code here. We are putting our `style.css` files under `/public/assets/css/`.

For performing Ajax requests, we need JavaScript coding. This code posts our `add_task` and `edit_task` forms, and updates them when our tasks are completed. Let's create a JavaScript file with the name `todo.js` in `/public/assets/js/`. The files contain the following code:

```
function task_done(id){

  $.get("/done/"+id, function(data) {

    if(data=="OK"){

      $("#"+id).addClass("done");
    }

  });
}
function delete_task(id){

  $.get("/delete/"+id, function(data) {
```

```
    if(data=="OK"){
      var target = $("#"+id);

      target.hide('slow', function(){ target.remove(); });

    }

  });
}


function show_form(form_id){

  $("form").hide();

  $('#'+form_id).show("slow");

}
function edit_task(id,title){

  $("#edit_task_id").val(id);

  $("#edit_task_title").val(title);

  show_form('edit_task');
}
$('#add_task').submit(function(event) {

  /* stop form from submitting normally */
  event.preventDefault();

  var title = $('#task_title').val();
  if(title){
    //ajax post the form
    $.post("/add", {title: title}).done(function(data) {

      $('#add_task').hide("slow");
      $("#task_list").append(data);
    });

  }
  else{
    alert("Please give a title to task");
    }
```

```
});

$('#edit_task').submit(function() {

  /* stop form from submitting normally */
  event.preventDefault();

  var task_id = $('#edit_task_id').val();
  var title = $('#edit_task_title').val();
  var current_title = $("#span_"+task_id).text();
  var new_title = current_title.replace(current_title, title);
  if(title){
    //ajax post the form
    $.post("/update/"+task_id, {title: title}).done(function(data)
      {
      $('#edit_task').hide("slow");
      $("#span_"+task_id).text(new_title);
    });
  }
  else{
    alert("Please give a title to task");
  }
});
```

Let's examine the JavaScript file.

Inserting data to the database with Ajax

In this application, we'll use the **Ajax POST** method for inserting data to the database. jQuery is the best JavaScript framework for these kinds of applications. jQuery also comes with powerful selector functions.

We have two forms in our HTML code, so we need to post them with Ajax to insert or update the data. We'll do it with jQuery's `post()` method.

We'll serve our JavaScript files under `/public/assets/js`, so let's create a `todo.js` file under this directory. First we need a request to add new tasks. The JavaScript code contains the following code:

```
$('#add_task').submit(function(event) {
  /* stop form from submitting normally */
  event.preventDefault();
  var title = $('#task_title').val();
  if(title){
    //ajax post the form
```

```
    $.post("/add", {title: title}).done(function(data) {
      $('#add_task').hide("slow");
      $("#task_list").append(data);
    });
  }
  else{
    alert("Please give a title to task");
  }
});
```

This code posts our `add_task` form to the server if the user remembers to provide a title to the task. If the user forgets to provide a title to the task, the code does not post the form. After it is posted, the code will hide the form and append the task list with a new record. Meanwhile, we will be waiting for the response to get the data.

So we need a second form to update a task's title. The code will update the task's title and change the text of updated records via Ajax on-the-fly. On-the-fly programming (or live coding) is a style of programming in which the programmer/performer/composer augments and modifies the program while it is running, without stopping or restarting, in order to assert expressive, programmable control for performance, composition, and experimentation at runtime. Because of the fundamental powers of programming languages, we believe the technical and aesthetic aspects of on-the-fly programming are worth exploring in web applications. The update form's code should be as follows:

```
$('#edit_task').submit(function(event) {
  /* stop form from submitting normally */
  event.preventDefault();
  var task_id = $('#edit_task_id').val();
  var title = $('#edit_task_title').val();
  var current_title = $("#span_"+task_id).text();
  var new_title = current_title.replace(current_title, title);
  if(title){
    //ajax post the form
    $.post("/update/"+task_id, {title: title}).done(function(data)
      {
      $('#edit_task').hide("slow");
      $("#span_"+task_id).text(new_title);
    });
  }
  else{
    alert("Please give a title to task");
  }
});
```

Laravel has the RESTful controller feature. This means you can define the RESTful base of the routes and controller functions. Also, routes can be defined for different request types such as **POST**, **GET**, **PUT**, or **DELETE**.

Before defining the routes, we need to code our controller. The controller files stay under `app/controllers/`; create a file in it named `TodoController.php`. The controller code should be as follows:

```
<?php
class TodoController extends BaseController
{
  public $restful = true;
  public function postAdd() {
    $todo = new Todo();
    $todo->title = Input::get("title");
    $todo->save();
    $last_todo = $todo->id;
    $todos = Todo::whereId($last_todo)->get();
    return View::make("ajaxData")
      ->with("todos", $todos);
  }
  public function postUpdate($id) {
    $task = Todo::find($id);
    $task->title = Input::get("title");
    $task->save();
    return "OK";
  }
}
```

Let's examine the code.

As you can see in the code, RESTful functions define syntaxes such as `postFunction`, `getFunction`, `putFunction`, or `deleteFunction`.

We have two post forms, so we need two POST functions and one GET method to get records from the database and show them in the template in the `foreach` statement to the visitor.

Let's examine the `postUpdate()` method in the preceding code:

```
public function postUpdate($id) {
  $task = Todo::find($id);
  $task->title = Input::get("title");
  $task->save();
  return "OK";
}
```

The following points explain the preceding code:

- The method needs a record called `id` to update. The route where we post would be similar to `/update/record_id`.
- `$task = Todo::find($id);` is that part of the method which finds the record from the database which has the given `id`.
- `$task->title = Input::get("title");` means to get the value of the form element named `title` and updating the `title` column record as the posted value.
- `$task->save();` applies the changes and runs the update query on the database server.

Let's examine the `postAdd()` method. This method works like our `getIndex()` method. The first part of the code creates a new record on the database server:

```
public function postAdd() {
  $todo = new Todo();
  $todo->title = Input::get("title");
  $todo->save();
  $last_todo = $todo->id;
  $todos = Todo::whereId($last_todo)->get();
  return View::make("ajaxData")
    ->with("todos", $todos);
}
```

The following points explain the preceding code:

- The code line `$last_todo = $todo->id;` gets the ID of this record. It is equivalent to the `mysql_insert_id()` function.
- The code line `$todos = Todo::whereId($last_todo)->get();` fetches the record from the `todo` table which has an `id` column equal to `$last_todo` variable.
- The code line `View::make("ajaxData") ->with("todos", $todos);` is very important to understand Laravel's view mechanism:
    - The code line `View::make("ajaxData")` refers to our template file. Do you remember the `ajaxData.blade.php` file, which we created under `/app/views/`? The code calls this file.
    - The code line `->with("todos", $todos);` assigns the last record to the template file as a variable named `todos` (the first parameter). So, we can show the last record in the template file with the `foreach` loop.

Retrieving the list from the database

We also need a method for getting the existing data from our database server. In our controller file, we need the function as shown in the following code:

```
public function getIndex() {
  $todos = Todo::all();
  return View::make("index")
    ->with("todos", $todos);
}
```

Let's examine the `getIndex()` method:

- In the code, `$todos = Todo:all()` means to get all records from the database and assign them to the `$todos` variable.
- In the code, `View::make("index")` defines our template file. Did you remember the `index.blade.php file`, which we created under `/app/views/`? The code calls this file.
- In the code, `->with("todos", $todos);` assigns the records to the template file. So, we can show the records in the template file with the `foreach` loop.

Finally, we will define our routes. For defining routes, you should open the `routes.php` file in the `apps` folder. Laravel has a great feature for defining routes named the RESTful controller. You can define all the routes with a single line of code as follows:

```
Route::controller('/', 'TodoController');
```

The preceding code assigns all the applications' root-based requests to the `TodoController` function. If you need to, you can also define the routes manually as follows:

```
Route::method('path/{variable}', 'TheController@functionName');
```

How to allow only Ajax requests

Our application accepts all POST and GET requests even without Ajax. But we just need to allow an Ajax request for `add` and `update` functions. Laravel's `Request` class provides many methods for examining the HTTP request for your applications. One of these functions is named `ajax()`. We can check the request type under controllers or route filters.

Allowing the request using route filters

Route filters provide a convenient way of limiting, accessing, or filtering the requests to a given route. There are several filters included in Laravel, which are located in the `filters.php` file in the `app` folder. We can define our custom filter under this file. We'll not use this method in this chapter, but we'll examine route filters in further chapters. The route filter for an Ajax request should be as shown in the following code:

```
Route::filter('ajax_check', function()
{
  if (Request::ajax())
    {
      return true;
    }
});
```

Attaching a filter to a route is also very easy. Check the sample route shown in the following code:

```
Route::get('/add', array('before' => 'ajax_check', function()
{
    return 'The Request is AJAX!';
}));
```

In the preceding example, we defined a route filter to the route with the `before` variable. This means, our application first checks the request type and then calls the controller function and passes the data.

Allowing the request using the controller side

We can check for the request type under controller. We'll use this method in this section. This method is useful for function-based filtering. For doing this, we should change our `add` and `update` functions as shown in the following code:

```
public function postAdd() {
  if(Request::ajax()){
    $todo = new Todo();
    $todo->title = Input::get("title");
    $todo->save();
    $last_todo = $todo->id;
    $todos = Todo::whereId($last_todo)->get();
    return View::make("ajaxData")
      ->with("todos", $todos);
  }
}
```

```
public function postUpdate($id) {
  if(Request::ajax()){
    $task = Todo::find($id);
    $task->title = Input::get("title");
    $task->save();
    return "OK";
  }
}
```

Wrapping up

In this chapter, we coded to add a new task, updated it, and listed the tasks. We also need to update each status and delete the tasks. For doing that, we need two functions that are called `getDone()` and `getDelete()`. As you know from previous sections of this chapter, these functions are RESTful and accept GET method requests. So, our function should be as shown in the following code:

```
public function getDelete($id) {
  if(Request::ajax()){
    $todo = Todo::whereId($id)->first();
    $todo->delete();
    return "OK";
  }
}
public function getDone($id) {
  if(Request::ajax()){
    $task = Todo::find($id);
    $task->status = 1;
    $task->save();
    return "OK";
  }
}
```

We also need to update the `todo.js` file. The final JavaScript code should be as shown in the following code:

```
function task_done(id){
  $.get("/done/"+id, function(data) {
    if(data=="OK"){
      $("#"+id).addClass("done");
    }
  });
}
function delete_task(id){
```

```
  $.get("/delete/"+id, function(data) {
    if(data=="OK"){
      var target = $("#"+id);
      target.hide('slow', function(){ target.remove(); });
    }
  });
}
function show_form(form_id){
  $("form").hide();
  $('#'+form_id).show("slow");
}
function edit_task(id,title){
  $("#edit_task_id").val(id);
  $("#edit_task_title").val(title);
  show_form('edit_task');
}
$('#add_task').submit(function(event) {
/* stop form from submitting normally */
  event.preventDefault();
  var title = $('#task_title').val();
  if(title){
    //ajax post the form
    $.post("/add", {title: title}).done(function(data) {
      $('#add_task').hide("slow");
      $("#task_list").append(data);
    });
  }
  else{
    alert("Please give a title to task");
  }
});
$('#edit_task').submit(function(event) {
/* stop form from submitting normally */
  event.preventDefault();
  var task_id = $('#edit_task_id').val();
  var title = $('#edit_task_title').val();
  var current_title = $("#span_"+task_id).text();
  var new_title = current_title.replace(current_title, title);
  if(title){
    //ajax post the form
    $.post("/update/"+task_id, {title:
      title}).done(function(data) {
      $('#edit_task').hide("slow");
      $("#span_"+task_id).text(new_title);
```

```
        });
      }
      else{
        alert("Please give a title to task");
      }
    });
```

Summary

In this section we tried to understand how to use Ajax with Laravel. Throughout the chapter, we used the basics of templating, request filtering, routing, and RESTful controllers. We also learned to update and delete data from our database.

In the next chapter we'll try to examine Laravel's file validation and file processing methods.

3
Building an Image Sharing Website

With this chapter, we are going to create a photo sharing website. First, we are going to create an images table. Then we'll cover methods to resize and share images.

The following topics are covered in this chapter:

- Creating a database and migrating the images table
- Creating a photo model
- Setting custom configuration values
- Installing a third-party library
- Creating a secure form for file upload
- Validating and processing the form
- Showing the image with a user interface
- Listing images
- Deleting the image from the database and server

Creating a database and migrating the images table

After successfully installing Laravel 4 and defining database credentials from `app/config/database.php`, create a database called `images`. For this, you can either create a new database from your hosting provider's panel, or if you are the server administrator, you can simply run the following SQL command:

```
CREATE DATABASE images
```

After successfully creating the database for the application, we need to create a `photos` table and install it to the database. To do this, open up your terminal, navigate to your project folder and run the following command:

```
php artisan migrate:make create_photos_table --table=photos –create
```

This command will generate a new MySQL database migration for us to create a table named photos.

Now we need to define what sections should be in our database table. For our example, I thought `id column`, `image titles`, `image file names`, and `timestamps` should be sufficient. So for this, open the migration file just created with the preceding command and change its contents as shown in the following code:

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

class CreatePhotosTable extends Migration {

  /**
  * Run the migrations.
  * @return void
  */
  public function up()
    {
    Schema::create('photos', function(Blueprint $table)
    {
      $table->increments('id');
      $table->string('title',400)->default('');
        //the column that holds the image's name
      $table->string('image',400)->default('');
        //the column that holds the image's filename
      $table->timestamps();
    });
  }

  /**
  * Reverse the migrations.
  * @return void
  */
  public function down()
  {
    Schema::drop('photos');
  }
}
```

After saving the file, run the following command to execute migrations:

```
php artsian migrate
```

If no error has occurred, you are ready for the next step of the project.

Creating a photo model

As you know, for anything related to database operations on Laravel, using models is the best practice. We will take advantage of the **Eloquent ORM**.

Save the following code as `images.php` in the `app/models/` directory:

```
<?php
class Photo extends Eloquent {

  //the variable that sets the table name
  protected $table = 'photos';

  //the variable that sets which columns can be edited
  protected $fillable = array('title','image');

  //The variable which enables or disables the Laravel's
    timestamps option. Default is true. We're leaving this here
    anyways
  public $timestamps = true;
}
```

We have set the table name with the `protected $table` variable. The content of which columns of the table can be updated/inserted will be decided with the `protected $fillable` variable. Finally, whether the model can add/update timestamps or not will be decided by the value of the `public $timestamps` variable. Just by setting this model (even without setting anything), we can easily use all the advantages of Eloquent ORM.

Our model is ready, now we can proceed to the next step and start to create our controller along with the upload forms. But before this, we are missing one simple thing. Where should the images be uploaded? What will be the maximum width and height of the thumbnails? To set these configuration values (think of it like constants of raw PHP), we should create a new configuration file.

Setting custom configuration values

With Laravel, setting configuration values is quite easy. All `config` values are within an array and will be defined as a `key=>value` pair.

Now let's make a new configuration file. Save this file as `image.php` in `app/config`:

```
<?php

/**
 * app/config/image.php
*/

return array(

  //the folder that will hold original uploaded images
  'upload_folder' => 'uploads',

  //the folder that will hold thumbnails
  'thumb_folder' => 'uploads/thumbs',

  //width of the resized thumbnail
  'thumb_width' => 320,

  //height of the resized thumbnail
  'thumb_height' => 240

);
```

You can set any other setting as you like. That's limited to your imagination. You can call the settings with Laravel's built-in `Config` Library's `get()` method. Sample usage is as shown in the following code:

```
Config::get('filename.key')
```

There is a dot (`.`) between the parameter, which splits the string into two. The first part is the filename of the `Config` without the extension, the second part is the key name of the configuration value. In our example, if we want to identify which folder is the uploaded folder name, we should write it as shown in the following code:

```
Config::get('image.upload_folder')
```

The preceding code will return whatever the value is. In our example, it will return `public/uploads`.

One more thing: We defined some folder names for our app, but we didn't create them. For some server configurations, the folders may be autocreated at the first attempt to upload the file, but if you don't create them, most probably it will cause errors on your server configuration. Create the following folders in the `public` folder and make them writable:

- `uploads/`
- `uploads/thumbs`

Now we should make an upload form for our image site.

Installing a third-party library

We should make an upload form and then a controller for our image site. But before doing that, we will install a third-party library for image processing as we will be benefiting from its methods. Laravel 4 uses **Composer**, so it's quite easy to install packages, update packages, and even update Laravel. For our project, we will be using a library called `Intervention`. The following steps must be followed to install the library:

1. First, make sure you have the latest `composer.phar` file by running `php composer.phar self-update` in your terminal.
2. Then open `composer.json` and add a new value to the `require` section. The value for our library is `intervention/image: "dev-master"`.

   Currently, our `composer.json` file's `require` section looks as follows:

   ```
   "require": {
     "laravel/framework": "4.0.*",
     "intervention/image": "dev-master"
   }
   ```

   You can find more packages for Composer at `www.packagist.org`.

3. After setting the value, open your terminal, navigate to the project's `root` folder, and type the following command:

   ```
   php composer.phar update
   ```

   This command will check `composer.json` and update all the dependencies (including Laravel itself) and if new requirements are added, it will download and install them.

4. After successfully downloading the library, we will now activate it. For this, we refer to the website of the `Intervention` class. Now open your `app/config/app.php`, and add the following value to the `providers` key:

   ```
   Intervention\Image\ImageServiceProvider
   ```

5. Now, we need to set an alias so that we can call the class easily. To do this, add the following value to the aliases key of the same file:

   ```
   'Image' => 'Intervention\Image\Facades\Image',
   ```

6. The class has a notation that is quite easy to understand. To resize an image, running the following code will suffice:

   ```
   Image::make(Input::file('photo')->getRealPath())
     ->resize(300, 200)->save('foo.jpg');
   ```

For more information about the `Intervention` class, go to the following web address:

`http://intervention.olivervogel.net`

Now, everything for the views and the form processing is ready; we can go on to the next step of our project.

Creating a secure form for file upload

Now we should make an upload form for our image site. We must make a view file, which will be loaded over a controller.

1. First, open up `app/routes.php`, remove the line starting with `Route::get()` that comes with Laravel, and add the following lines:

   ```
   //This is for the get event of the index page
   Route::get('/',array('as'=>'index_page','uses'=>
     'ImageController@getIndex'));
   //This is for the post event of the index.page
   Route::post('/',array('as'=>'index_page_post','before' =>
     'csrf', 'uses'=>'ImageController@postIndex'));
   ```

   The key `'as'` defines the name of the route (like a shortcut). So if you make links to the routes, even if the URL changes for the route, your links to the application won't be broken. The `before` key defines what filters will be used before the action starts. You can define your own filters, or use the built-in ones. We set `csrf`, so the **CSRF** (**Cross-site Request Forgery**) checking will be done before the action starts. This way, you can prevent attackers from injecting unauthorized requests into your application. You can use multiple filters with the separator; for example, `filter1|filter2`.

You can also define the CSRF protection from controllers directly.

2. Now, let's create our first method for the controller. Add a new file containing the following code and name it `ImageController.php` in `app/controllers/`:

```
<?php

class ImageController extends BaseController {


  public function getIndex()
  {
    //Let's load the form view
    return View::make('tpl.index');
  }

}
```

Our controller is RESTful; that's why our method index is named `getIndex()`. In this method, we are simply loading a view.

3. Now let's create a master page for the view using the following code. Save this file as `frontend_master.blade.php` in `app/views/`:

```
<!DOCTYPE HTML PUBLIC "-//W3C//
  DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<html lang="en">
  <head>
  <meta http-equiv="content-type"
    content="text/html; charset=utf-8">
  <title>Laravel Image Sharing</title>
  {{HTML::style('css/styles.css')}}
  </head>

  <body>
    {{--Your title of the image (and yeah, blade engine
      has its own commenting, cool, isn't it?)--}}
    <h2>Your Awesome Image Sharing Website</h2>

    {{--If there is an error flashdata in session
      (from form validation), we show the first one--}}
```

```
    @if(Session::has('errors'))
      <h3 class="error">{{$errors->first()}}</h3>
    @endif

    {{--If there is an error flashdata in session which
      is set manually, we will show it--}}
    @if(Session::has('error'))
      <h3 class="error">{{Session::get('error')}}</h3>
    @endif

    {{--If we have a success message to show, we print
      it--}}
    @if(Session::has('success'))
      <h3 class="error">{{Session::get('success')}}</h3>
    @endif

    {{--We yield (get the contents of) the section named
      'content' from the view files--}}
    @yield('content')

  </body>
</html>
```

To add a `CSS` file (which we will create in the next steps), we use the `style()` method of the `HTML` class. And our masterpage yields a section named `content`, which will be filled with the `view files` sections.

4. Now, let's create our `view file` section by using the following code. Save this file as `index.blade.php` in the `app/views/tpl/`directory:

```
@extends('frontend_master')

@section('content')
  {{Form::open(array('url' => '/', 'files' => true))}}
  {{Form::text('title','',array('placeholder'=>
    'Please insert your title here'))}}
  {{Form::file('image')}}
  {{Form::submit('save!',array('name'=>'send'))}}
  {{Form::close()}}
@stop
```

In the first line of the preceding code, we told the Blade Engine that we will be using `frontend_master.blade.php` as the layout. This is done using the `@extends()` method in Laravel 4.

If you are coming from Laravel 3, `@layout` is renamed as `@extends`.

Benefiting from the `Form` class of Laravel, we generated an upload form with the `title` field and `upload` field. Unlike Laravel 3, to make a new upload form, we are not using `Form::open_for_files()` anymore. It's merged with the `open()` method, which accepts either a string or an array if you want to pass more than one parameter. We will be passing the action URL as well as telling it that it's an upload form, so we passed two parameters. The `url` key is to define where the form will be submitted. The `files` parameter is Boolean, and if it's set to `true`, it'll make the form an upload form, so we can work with files.

To secure the form and to prevent unwanted form submission attempts, we will be needing a CSRF key `hidden` in our form. Thanks to Laravel's `Form` class, it's generated in the form automatically, right after the form opening tag. You can check it by looking at the source code of the generated form.

The hidden autogenerated CSRF form element looks as follows:

```
<input name="_token" type="hidden" value="SnRocsQQlOnqEDH45ewP2GLx
PFUy5eH4RyLzeKm3">
```

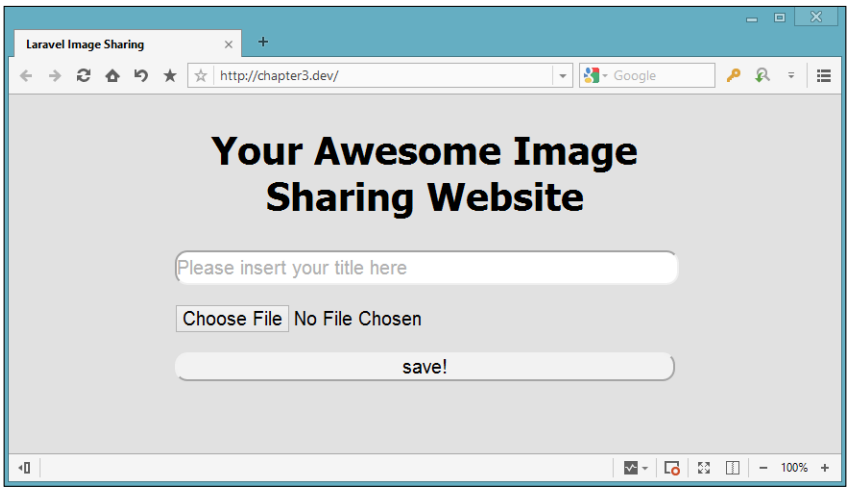
5. Now let's tidy up the form a bit. This is not directly related to our project, but just for the look. Save the `styles.css` file in `public/css/` (the path we defined on the master page):

```
/*Body adjustment*/
body{width:60%; margin:auto; background:#dedede}
/*The title*/
h2{font-size:40px; text-align:center; font-family:
  Tahoma,Arial,sans-serif}
/*Sub title (success and error messages)*/
h3{font-size:25px; border-radius:4px; font-family:
  Tahoma,Arial,sans-serif; text-align:center;
  width:100%}
h3.error{border:3px solid #d00; background-color:
  #f66; color:#d00 }
h3.success{border:3px solid #0d0; background-color:
  #0f0; color:#0d0}
p{font-size:25px; font-weight: bold; color: black;
  font-family: Tahoma,Arial,sans-serif}
ul{float:left;width:100%;list-style:none}
  li{float:left;margin-right:10px}
```

```
/*For the input files of the form*/
input{float:left; width:100%; border-radius:13px;
  font-size:20px; height:30px; border:10px 0 10px 0;
  margin-bottom:20px}
```

We've styled the body by giving it 60 percent width, making it center-aligned, and giving it a grayish background. We also formatted `h2` and `h3` messages with `success` and `error` classes, and `forms`.

After the styling, the form will look as shown in the following screenshot:



Now that our form is ready, we are ready to progress to the next step of the project.

Validating and processing the form

In this section, we are going to validate the submitted form and make sure that the required fields are present and the submitted file is an image. Then we will upload the image to our server, process the image, create the thumbnail, and save the image information to the database as follows:

1. First, we need to define the form validation rules. We prefer adding such values to the related model, so the rules become reusable, and this prevents the code from becoming bloated. To do this, add the following code in the `photo.php` file in the `app/models/` directory (the model that we generated earlier in this chapter) inside the class definition before the last curly bracket(}):

```
//rules of the image upload form
public static $upload_rules = array(
```

```
  'title'=> 'required|min:3',
  'image'=> 'required|image'
);
```

We set the variable as `public`, so it can be used outside the model file, and we set it to static, so we can directly access the variable.

We want both `title` and `image` to be mandatory, and `title` should have at least three characters. Also, we want to check MIME types of the `image` column and make sure that it's an image.

Laravel's MIME-type checking requires the `Fileinfo` extension to be installed. So make sure it's enabled in your PHP configuration.

2. Now we need the controller's `post` method to process the form. Add this method in the `ImageController.php` file in `app/controllers/` before the last curly bracket(}):

```
public function postIndex()
{

  //Let's validate the form first with the rules which are
    set at the model
  $validation = Validator::make(Input::all(),
    Photo::$upload_rules);

  //If the validation fails, we redirect the user to the
    index page, with the error messages
  if($validation->fails()) {
    return Redirect::to('/')
      ->withInput()
      ->withErrors($validation);
  }
  else {

    //If the validation passes, we upload the image to the
      database and process it
    $image = Input::file('image');

    //This is the original uploaded client name of the
      image
    $filename = $image->getClientOriginalName();
    //Because Symfony API does not provide filename
      //without extension, we will be using raw PHP here
```

```
$filename = pathinfo($filename, PATHINFO_FILENAME);

//We should salt and make an url-friendly version of
  //the filename
//(In ideal application, you should check the filename
  //to be unique)
$fullname = Str::slug(Str::random(8).$filename).'.'.
  $image->getClientOriginalExtension();

//We upload the image first to the upload folder, then
  get make a thumbnail from the uploaded image
$upload = $image->move
  (Config::get( 'image.upload_folder'),$fullname);

//Our model that we've created is named Photo, this
  library has an alias named Image, don't mix them two!
//These parameters are related to the image processing
  class that we've included, not really related to
  Laravel
Image::make(Config::
  get( 'image.upload_folder').'/'.$fullname)
  ->resize(Config::get( 'image.thumb_width'),
  null, true)
  ->save(Config::get
  ( 'image.thumb_folder').'/'.$fullname);

//If the file is now uploaded, we show an error message
  to the user, else we add a new column to the database
  and show the success message
if($upload) {

  //image is now uploaded, we first need to add column
    to the database
  $insert_id = DB::table('photos')->insertGetId(
    array(
      'title' => Input::get('title'),
      'image' => $fullname
    )
  );

  //Now we redirect to the image's permalink
  return Redirect::to(URL::to('snatch/'.$insert_id))
    ->with('success','Your image is uploaded
    successfully!');
```

```
            } else {
              //image cannot be uploaded
              return Redirect::to('/')->withInput()
                ->with('error','Sorry, the image could not be
                uploaded, please try again later');
            }
          }
        }
```

Let's dig the code one by one.

1. First, we made a form validation and called our validation rules from the model that we've generated via `Photo::$upload_rules`.
2. Then we've salted (added additional random characters for security) the filename and made the filename URL-friendly. First, we get the uploaded filename with the getClientOriginalName() method, then get the extension with the getClientOriginalExtension() method. We salted the filename with an eight character-long random string, which we gained by the random() method of the STR class. Lastly, we made the filename URL-friendly with Laravel's built-in slug() method of the STR class.
3. After all the variables are ready, we first uploaded the file to the server with the move() method, which takes two parameters. The first parameter is the path to which the file is going to be transferred, the second parameter is the filename of the uploaded file.
4. After uploading, we created a static thumbnail for the uploaded image. For this, we benefited from Intervention, an image processing class we've implemented earlier.
5. Lastly, if everything goes okay, we add the title and image filenames to the database and get the ID with the insertGetId() method of Fluent Query Builder, which inserts the row first and returns insert_id of the column. We could also create the row with Eloquent ORM by setting the create() method to a variable and get the id_column name such as $create->id.
6. After everything is okay and we get `insert_id`, we redirect the user to a new page that will show thumbnails, full-image links, and a forum thumbnail **BBCode**, which we will generate in the next sections.

Showing the image with a user interface

Now, we need to make a new view and method from the controller to show the information of the image uploaded. This can be done as follows:

1. First, we need to define a `GET` route for the controller. For this, open your file `routes.php` in the `app` folder and add the following codes:

```
//This is to show the image's permalink on our website
Route::get('snatch/{id}',
  array('as'=>'get_image_information',
  'uses'=>'ImageController@getSnatch'))
  ->where('id', '[0-9]+');
```

   We defined an `id` variable on the route, and with the `where()` method, using regular expression, we filtered it first hand. So we don't need to worry about filtering the ID field, whether it's a natural number or not.

2. Now, let's create our controller method. Add the following code inside `ImageController.php` in `app/controllers/` before the last curly bracket (}):

```
public function getSnatch($id) {
  //Let's try to find the image from database first
  $image = Photo::find($id);
  //If found, we load the view and pass the image info as
    parameter, else we redirect to main page with error
    message
  if($image) {
    return View::make('tpl.permalink')
      ->with('image',$image);
  } else {
    return Redirect::
      to('/')->with('error','Image not found');
  }
}
```

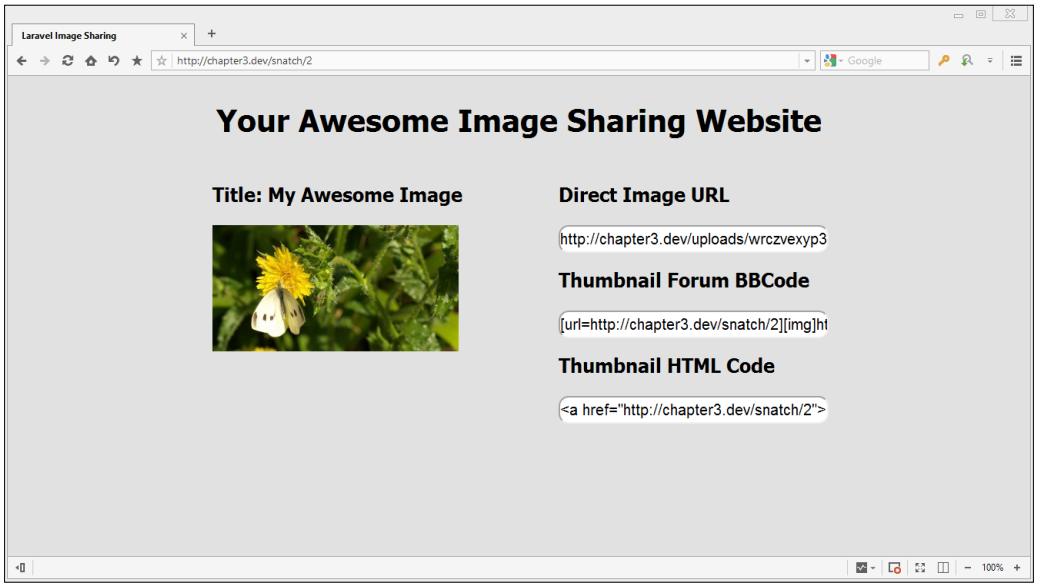
   First, we looked for the image with the `find()` method of Eloquent ORM. If it returns the value as false, that means there is a row found. So we can simply check whether there is a result or not with a simple `if` clause. If there is a result, we will load our view with the found image info as a variable named `$image`, using the `with()` method. If no values are found, we return to the index page with an error message.

3. Now let's create the template file containing the following code. Save this file as `permalink.blade.php` in `app/views/tpl/`:

```
@extends('frontend_master')
@section('content')
<table cellpadding="0" cellspacing="0" border="0"
width="100percent">
  <tr>
    <td width="450" valign="top">
      <p>Title: {{$image->title}}</p>
    {{HTML::image(Config::get
      ('image.thumb_folder').'/'.$image->image)}}
    </td>
      <td valign="top">
      <p>Direct Image URL</p>
      <input onclick="this.select()" type="text"
        width="100percent" value=
        "{{URL::to(Config::get('image.upload_folder').'/'
        $image->image)}}" />

      <p>Thumbnail Forum BBCode</p>
      <input onclick="this.select()" type="text"
        width="100percent" value=
        "[url={{URL::to('snatch/'
        $image->id)}}][img]{{
        URL::to(Config::get('image.thumb_folder')
        '/'.$image->image)}}[/img][/url]" />

      <p>Thumbnail HTML Code</p>
      <input onclick="this.select()" type="text"
        width="100percent"
        value="{{HTML::entities(HTML::link(
        URL::to('snatch/'.$image->id),
        HTML::image(Config::get('image.thumb_folder').'/'
        $image->image)))}}" />
    </td>
  </tr>
</table>
@stop
```

You should be familiar with most methods used in this template by now. There is a new method called `entities()` of the `HTML` class, which actually is `htmlentities()` of raw PHP, but with some pre-checks and as Laravel's way.

Also, because we've returned the `$image` variable to the view (which is the database row object that we've gained using Eloquent), we can use it directly as `$image->columnName` in the view.
This will produce a view as shown in the following image:



4. We have added a permalink feature for our project, but what if we want to show all the images? For that, we need an `'all pages'` section in our system.

Listing images

In this section, we are going to create an `'all images'` section in our system, which will have a page navigation (pagination) system. There are a few steps to be followed as shown:

1. First, we need to define its URL from our `route.php` file. For this, open `app/routes.php` and add the following lines:

```
//This route is to show all images.
Route::get('all',array('as'=>'all_images','uses'=>'ImageController@
getAll'));
```

2. Now, we need a method named `getAll()` (there is a `get` method at the start because it will be a RESTful controller) to get values and load the view. To do this, open your `app/controllers/ImageController.php` and add these codes before the last curly bracket (}):

```
public function getAll(){

  //Let's first take all images with a pagination feature
  $all_images = DB::
    table('photos')->orderBy('id','desc')->paginate(6);

  //Then let's load the view with found data and pass the
    variable to the view
  return View::make('tpl.all_images')
    ->with('images',$all_images);
}
```

   Here, we first got all the images from the database using the `paginate()` method, which will allow us to get the pagination links easily. After that, we loaded the view for the user with the images data with pagination.

3. To view this properly, we need a view file. Save the following code in a file named `all_image.blade.php` in the `app/views/tpl/` directory:

```
@extends('frontend_master')

@section('content')

@if(count($images))
  <ul>

    @foreach($images as $each)
      <li>
        <a href="{{URL::to('snatch/'
          $each->id)}}">{{
          HTML::image(Config::get('image.thumb_folder')
          '/'.$each->image)}}</a>
      </li>
    @endforeach
  </ul>
  <p>{{$images->links()}}</p>
@else
  {{--If no images are found on the database, we will show
    a no image found error message--}}
  <p>No images uploaded yet, {
    {HTML::link('/','care to upload one?')}}</p>
@endif
@stop
```

We first extend the `frontend_master.blade.php` file with our content section. As for the content section, we first check whether any rows are returned. If so, then we loop them all in list item tags (`<li>`) with their permalinks. The `links()` method that came with the `paginate` class will create the pagination for us.

You can switch the pagination template from `app/config/view.php`.

If no rows have returned, that means there are no images (yet), so we show a warning message with a link to the new upload page (which is the index page in our case).

What if a person uploads an image that is not allowed or not safe for work? You would not like to have them on your website, right? So there should be an image deleting feature on your website.

Deleting the image from the database and server

We would like to have a delete feature in our script, using which we will delete the image both from the database and from its uploaded folder. This process is quite easy with Laravel.

1. First, we need to create a new route for the action. To do this, open `app/routes.php` and add the following lines:

```
//This route is to delete the image with given ID
Route::get('delete/{id}', array
  ('as'=>'delete_image','uses'=>
  'ImageController@getDelete'))
    ->where('id', '[0-9]+');
```

2. Now, we need to define the controller method `getDelete($id)` inside `ImageController`. To do this, open `app/controllers/ImageController.php` and add the following code above the last curly bracket (}):

```
public function getDelete($id) {
  //Let's first find the image
  $image = Photo::find($id);

  //If there's an image, we will continue to the deleting
    process
  if($image) {
```

```
        //First, let's delete the images from FTP
        File::delete(Config::get('image.upload_folder').'/'
          $image->image);
        File::delete(Config::get('image.thumb_folder').'/'
          $image->image);

        //Now let's delete the value from database
        $image->delete();

        //Let's return to the main page with a success message
        return Redirect::to('/')
          ->with('success','Image deleted successfully');

      } else {
        //Image not found, so we will redirect to the index
          page with an error message flash data.
        return Redirect::to('/')
          ->with('error','No image with given ID found');
      }
    }
```

Let's understand the code:

1. First, we look at our database, and if we have an image with a given ID already with the `find()` method of Eloquent ORM, we will store it with a variable called `$image`.
2. If the value of the `$image` is not false, there is an image matching the image in our database. Then, we delete the file with the `delete()` method of the File class. Alternatively, you can also use the unlink() method of raw PHP.
3. After the files are deleted from the file server, we delete the image's information row from the database. To do this, we are using the `delete()` method of Eloquent ORM.
4. If everything goes smoothly, we should redirect back to the main page with a success message saying the image is deleted successfully.

In practical application, you should have a backend interface for such actions.

Summary

In this chapter, we've created a simple image sharing website with Laravel's built-in functions. We've learned how to validate our forms, how to work with files and check their MIME types, and set custom configuration values. We've learned more about database methods both with Fluent and Eloquent ORM. Also, for image processing, we've installed a third-party library from `packagist.org` using Composer and learned how to update them. We've also listed images with the page navigation feature and learned to delete files from the server. In the next chapter, we will be building a personal blog site with authentication and a members-only area, and we will assign blog posts to the author(s).

4
Building a Personal Blog

In this chapter, we'll code a simple personal blog with Laravel. We'll also cover Laravel's built-in authentication, paginate mechanism, and named routes. We'll elaborate some rapid development methods, which come with Laravel, such as creating route URLs. The following topics will be covered in this chapter:

- Creating and migrating the posts database
- Creating a posts model
- Creating and migrating the authors database
- Creating a members-only area
- Saving a blog post
- Assigning blog posts to users
- Listing articles
- Paginating the content

Creating and migrating the posts database

We assume that you have already defined database credentials in the `app/config/database.php` file. For this application, we need a database. You can simply create and run the following SQL command or basically you can use your database administration interface, something like phpMyAdmin:

```
CREATE DATABASE laravel_blog
```

After successfully creating the database for the application, first we need to create a posts table and install it in the database. To do this, open up your terminal, navigate through your project folder, and run this command:

```
php artisan migrate:make create_posts_table --table=posts --create
```

This command will generate a migration file under `app/database/migrations` for generating a new MySQL table named `posts` in our `laravel_blog` database.

To define our table columns and specifications, we need to edit this file. After editing the migration file, it should look like this:

```
<?php

use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreatePostsTable extends Migration {

  /**
   * Run the migrations.
   *
   * @return void
   */
  public function up()
  {
    Schema::create('posts', function(Blueprint $table)
    {
      $table->increments('id');
      $table->string('title');
      $table->text('content');
      $table->integer('author_id');
      $table->timestamps();
    });
  }

  /**
   * Reverse the migrations.
   *
   * @return void
   */
  public function down()
  {
    Schema::drop('posts');
  }
}
```

After saving the file, we need to use a simple `artisan` command to execute migrations:

```
php artisian migrate
```

If no error occurs, check the `laravel_blog` database for the `posts` table and columns.

Creating a posts model

As you know, for anything related to database operations on Laravel, using models is the best practice. We will benefit from the Eloquent ORM.

Save this code in a file named as `Posts.php` under `app/models/`:

```
<?php
class Post extends Eloquent {

protected $table = 'posts';

protected $fillable = array('title','content','author_id');

public $timestamps = true;

public function Author(){

      return $this->belongsTo('User','author_id');
}

}
```

We have set the database table name with the protected `$table` variable. We have also set editable columns with the `$fillable` variable and timestamps with the `$timestamps` variable as we've already seen and used in previous chapters. The variables which are defined in the model are enough for using Laravel's Eloquent ORM. We'll cover the public `Author()` function in the *Assigning blog posts to users* section of this chapter.

Our posts model is ready. Now we need an authors model and database to assign blog posts to authors. Let's investigate Laravel's built-in authentication mechanism.

Creating and migrating the authors database

Contrary to most of the PHP frameworks, Laravel has a basic authentication class. The authentication class is very helpful in rapidly developing applications. First, we need a secret key for our application. The application secret key is very important for our application's security because all data is hashed salting this key. The artisan command can generate this key for us with a single command line:

```
php artisian key:generate
```

If no error occurs, you will see a message that tells you that the key is generated successfully. After key generation, if you face problems with opening your Laravel application, simply clear your browser cache and try again. Next, we should edit the authentication class's configuration file. For using Laravel's built-in authentication class, we need to edit the configuration file, which is located at `app/config/auth.php`. The file contains several options for the authentication facilities. If you need to change the table name, and so on, you can make the changes under this file. By default, Laravel comes with the `User` model. You can see the `User.php` file, which is located at `app/models/`. With Laravel 4, we need to define which fields are fillable in our `Users` model. Let's edit `User.php` located at `app/models/` and add the "fillable" array:

```
<?php

use Illuminate\Auth\UserInterface;
use Illuminate\Auth\Reminders\RemindableInterface;

class User extends Eloquent implements UserInterface,
RemindableInterface {

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

  /**
   * The attributes excluded from the model's JSON form.
   *
   * @var array
   */
  protected $hidden = array('password');
```

```
    //Add to the "fillable" array
     protected $fillable = array('email', 'password', 'name');

    /**
     * Get the unique identifier for the user.
     *
     * @return mixed
     */
    public function getAuthIdentifier()
    {
      return $this->getKey();
    }

    /**
     * Get the password for the user.
     *
     * @return string
     */
    public function getAuthPassword()
    {
      return $this->password;
    }

    /**
     * Get the e-mail address where password reminders are sent.
     *
     * @return string
     */
    public function getReminderEmail()
    {
      return $this->email;
    }

}
```

Basically, we need three columns for our authors. These are:

- `email`: This column stores author's e-mails
- `password`: This column stores authors' passwords
- `name`: This column stores the authors' names and surnames

Now we need several migration files to create the `users` table and add an author to our database. To create a migration file, give a command such as the following:

```
php artisan migrate:make create_users_table --table=users --create
```

Open the migration file, which was created recently and located at `app/database/migrations/`. We need to edit the `up()` function as the following:

```
public function up()
{
  Schema::create('users', function(Blueprint $table)
  {
    $table->increments('id');
    $table->string('email');
    $table->string('password');
    $table->string('name');
    $table->timestamps();
  });
}
```

After editing the migration file, run the `migrate` command:

```
php artisian migrate
```

As you know, the command creates the `users` table and its columns. If no error occurs, check the `laravel_blog` database for the `users` table and the columns.

Now we need to create a new migration file for adding some authors to the database. We can do so by running the following command:

```
php artisan migrate:make add_some_users
```

Open up the migration file and edit the `up()` function as the following:

```
public function up()
{
  User::create(array(
          'email' => 'your@email.com',
          'password' => Hash::make('password'),
          'name' => 'John Doe'
      ));
}
```

We used a new class in the `up()` function, which is named `Hash`. Laravel has a hash maker/checker class, which is based on secure **Bcrypt**. Bcrypt is an accepted, secure hashing method for important data such as passwords.

The class for which we have created an application key with the artisan tool at the beginning of this chapter is used for salting. So, to apply migration, we need to migrate with the following artisan command:

```
php artisian migrate
```

Now, check the `users` table for a record. If you check the `password` column, you will see a record stored as follows:

```
$2y$08$ayylAhkVNCnkfj2rITbQr.L5pd2AIfpeccdnW6.BGbA.1VtJ6Sdqy
```

It is very important to securely store your user's passwords and their critical data. Do not forget that if you change the application key, all the existing hashed records will be unusable because the `Hash` class uses the application key as the salting key when validating and storing given data.

Creating a members-only area

As you know, our blog system is member based. Because of that we need some areas to be accessible by members only, for adding new posts to the blog. We have two different methods to do this. The first one is the route filter method, which we will elaborate in the next chapters. The second is the template-based authorization check. This method is a more effective way of understanding the use of the `Auth` class with the **Blade template system**.

With the `Auth` class we can check the authorization status of a visitor by just a single line of code:

```
Auth::check();
```

The `check()` function, which is based on the `Auth` class, always returns `true` or `false`. So, that means we can easily use the function in an `if/else` statement in our code. As you know from previous chapters, with the blade template system we were able to use that kind of PHP statement in the template files.

Before creating the template files we need to write our routes. We need four routes for our application. These are:

- A login route to process login requests
- A new post route to process new post requests
- An admin route to show a new post form and a login form
- An index route to list posts

Named routing is another amazing feature of the Laravel framework for rapid development. Named routes allow referring to routes when generating redirects or URLs more comfortably. You may specify a name for a route as follows:

```
Route::get('all/posts', array('as' => 'posts', function()
{
    //
}));
```

You may also specify route names for controllers:

```
Route::get('all/posts', array('as' => 'allposts', , 'uses' =>
'PostController@showPosts'));
```

Thanks to the named routes, we can easily create URLs for our application:

```
$url = URL::route('allposts');
```

We can also use the named routes to redirect:

```
$redirect = Redirect::route('allposts');
```

Open the route configuration file, which is located at `app/routes.php` and add the following code:

```
Route::get('/', array('as' => 'index', 'uses' => 'PostsController@
getIndex'));
Route::get('/admin', array('as' => 'admin_area', 'uses' =>
'PostsController@getAdmin'));
Route::post('/add', array('as' => 'add_new_post', 'uses' =>
'PostsController@postAdd'));
Route::post('/login', array('as' => 'login', 'uses' =>
'UsersController@postLogin'));
Route::get('/logout', array('as' => 'logout', 'uses' =>
'UsersController@getLogout'));
```

Now we need to write the code for the controller side and templates of our application. First, we can start coding from our admin area. Let's create a file under `app/views/` with the name `addpost.blade.php`. Our admin template should look like the following:

```
<html>
<head>
<title>Welcome to Your Blog</title>
<link rel="stylesheet" type="text/css" href="/assets/css/style.css">
<!--[if lt IE 9]><script src="//html5shim.googlecode.com/svn/trunk/
html5.js"></script><![endif]-->
</head>
```

```
<body>
@if(Auth::check())
<section class="container">
<div class="content">
<h1>Welcome to Admin Area, {{Auth::user()->name}} ! - <b>{{link_to_
route('logout','Logout')}}</b></h1>
<form name="add_post" method="POST" action="{{URL::route('add_new_
post')}}">
<p><input type="text" name="title" placeholder="Post Title"
value=""/></p>
<p><textarea name="content" placeholder="Post Content"></textarea></p>
<p><input type="submit" name="submit" /></p>
</div>
</section>
@else
<section class="container">
<div class="login">
<h1>Please Login</h1>
<form name="login" method="POST" action="{{URL::route('login')}}">
<p><input type="text" name="email" value="" placeholder="Email"></p>
<p><input type="password" name="password" value=""
placeholder="Password"></p>
<p class="submit"><input type="submit" name="commit" value="Login"></
p>
</form>
</div>
</section>
@endif
</body>
</html>
```

As you can see in the code, we use `if/else` statements in a template to check a user's login credentials. We know already from the beginning of this section that we use the `Auth::check()` function to check the login status of a user. Also, we've used a new method to get the currently logged in user's name:

```
Auth::user()->name;
```

We can get any information about the current user with the `user` method:

```
Auth::user()->id;
Auth::user()->email;
```

The template code first checks the login status of the visitor. If the visitor has logged in, the template shows a new post form; else it shows a login form.

Now we have to code the controller side of our blog application. Let's start from our users controller. Create a file under `app/controller/`, which is named `UsersContoller.php`. The final code of the controller should be as follows:

```
<?php

class UsersController extends BaseController{

  public function postLogin()
  {
    Auth::attempt(array('email' => Input::get('email'),
      'password' => Input::get('password')));
  return Redirect::route('add_new_post');

  }

  public function getLogout()
  {
    Auth::logout();
    return Redirect::route('index');
  }
}
```

The controller has two functions: the first is the `postLogin()` function. This function basically checks the posted form data for user login and then redirects the visitor to the `add_new_post` route to show the new post form. The second function processes the logout request and redirects to the `index` route.

Saving a blog post

Now we need one more controller for our blog posts. So, create a file under `app/controller/`, that is named `PostsContoller.php`. The final code of the controller should be as follows:

```
<?php
class PostsController extends BaseController{

  public function getIndex()
  {

  $posts = Post::with('Author')-> orderBy('id', 'DESC')->get();
  return View::make('index')->with('posts',$posts);

  }
```

```
  public function getAdmin()
  {
  return View::make('addpost');
  }
  public function postAdd()
  {
  Post::create(array(
              'title' => Input::get('title'),
              'content' => Input::get('content'),
              'author_id' => Auth::user()->id
   ));
  return Redirect::route('index');
  }
}
```

Assigning blog posts to users

The `postAdd()` function processes the new blog post create request on the database. As you can see, we can get the author's ID with a previously mentioned method:

```
Auth::user()->id
```

With this method, we can assign the current user with a blog post. As you will see, we have a new method in the query:

```
Post::with('Author')->
```

If you remember, we've defined a public `Author ()` function in our `Posts` model:

```
public function Author(){

      return $this->belongsTo('User','author_id');
}
```

The `belongsTo()` method is an `Eloquent` function to create relations between tables. Basically the function needs one required variable and one optional variable. The first variable (required) defines the target `Model`. The second and optional variable is to define the source column of the current model's table. If you don't define the optional variable, the `Eloquent` class searches the `targetModelName_id` column. In the `posts` table, we store the authors' IDs in the `author_id` column, not in the column named `user_id`. Because of this, we need to define a second optional variable in the function. With this method, we can pass our blog posts and all its authors' information to the template file. You can think of the method as some kind of a SQL join method.

When we want to use these relation functions in queries, we can easily call them as follows:

```
Books::with('Categories')->with('Author')->get();
```

It is easy to manage the template files with fewer variables. Now we have just one variable to pass the template file, which is combined with all the necessary data. So, we need the second template file to list our blog posts. This template will work at our blog's frontend.

Listing articles

In the previous sections of this chapter, we've learned to use PHP `if/else` statements within blade template files. Laravel passes data to the template file as an array. So we need to use the `foreach` loop to parse data into the template file. We can also use a `foreach` loop in template files. So create a file under `app/views/` named `index.blade.php`. The code should look as follows:

```
<!doctype html>
<html lang="en">
<head>
<meta charset="utf-8" />
<title>My Awesome Blog</title>
<link rel="stylesheet" href="/assets/blog/css/styles.css" type="text/
css" media="screen" />
<link rel="stylesheet" type="text/css" href="/assets/blog/css/print.
css" media="print" />
<!--[if IE]><script src="http://html5shiv.googlecode.com/svn/trunk/
html5.js"></script><![endif]-->
</head>
<body>
<div id="wrapper">
<header>
<h1><a href="/">My Awesome Blog</a></h1>
<p>Welcome to my awesome blog</p>
</header>
<section id="main">
<section id="content">
@foreach($posts as $post)
<article>
<h2>{{$post->title}}</h2>
```

```
<p>{{$post->content}}</p>
<p><small>Posted by <b>{{$post->Author->name}}</b> at <b>{{$post-
>created_at}}</b></small></p>
</article>

@endforeach
</section>
</aside>
</section>
<footer>
<section id="footer-area">
<section id="footer-outer-block">
<aside class="footer-segment">
<h4>My Awesome Blog</h4>
</aside>
</section>
</section>
</footer>
</div>
</body>
</html>
```

Let's dig the code. We've used a `foreach` loop inside the template file to parse all blog post data. Also, we see the combined author data usage in the `foreach` loop. As you may remember, we get the author information with the `belongsTo()` method in the model side. The whole relational data parsing is done inside an array, which is named the relation function name. For example, if we had a second relation function, which is named `Categories()`, the query would be something as follows on the controller side:

```
$books = Books::with('Author')-> with('Categories')->orderBy('id',
'DESC')->get();
```

The `foreach` loop would look as follows:

```
@foreach($books as $book)

<article>
<h2>{{$book->title}}</h2>
<p>Author: <b>{{$book->Author->name}}</b></p>
<p>Category: <b>{{$book->Category->name}}</b></p>
</article>

@endforeach
```

Paginating the content

Eloquent's `get()` method, which we've used in the controller side in the `Eloquent` query, fetches all the data from the database with a given condition. Often we need to paginate the content for a user-friendly frontend or less page loads and optimizations. The `Eloquent` class has a helpful method to do this quickly, which is called `paginate()`. This method fetches the data paginated and generates paginate links in the template with just a single line of code. Open the `app/controllers/PostsController.php` file and change the query as follows:

```
$posts = Post::with('Author')->orderBy('id', 'DESC')->paginate(5);
```

The `paginate()` method paginates the data with the given numeric value. So, the blog posts will be paginating each page into `5` blog posts. We have to also change our template to show pagination links. Open `app/views/index.blade.php` and add the following code after the `foreach` loop:

```
{{$posts->links()}}
```

The section in the template, which has the İD as `"main"`, should look as follows:

```
<section id="main">
<section id="content">
@foreach($posts as $post)

<article>
<h2>{{$post->title}}</h2>
<p>{{$post->content}}</p>
<p><small>Posted by <b>{{$post->Author->name}}</b> at <b>{{$post-
>created_at}}</b></small></p>
</article>
@endforeach

</section>
{{$posts->links()}}
</section>
```

The `links()` function will generate pagination links automatically, if there is enough data to paginate. Else, the function shows nothing.

Summary

In this chapter, we've created a simple blog with Laravel's built-in functions and the Eloquent database driver. We've learned how to paginate the data and Eloquent's basic data relation mechanism. Also we've covered Laravel's built-in authentication mechanism. In the next chapters, we'll learn how to work with more complex tables and relational data.

5
Building a News Aggregation Website

During this chapter, we will create a news aggregation site. We will parse multiple feeds, categorize them, activate/deactivate them for our website, and display them on our website using PHP's SimpleXML extension. The following topics will be covered in this chapter:

- Creating the database and migrating the feeds table
- Creating a feeds model
- Creating our form
- Validating and processing the form
- Extending the core classes
- Reading and parsing an external feed

Creating the database and migrating the feeds table

After successfully installing Laravel 4 and defining database credentials from `app/config/database.php`, create a database called `feeds`.

After creating the database, open your terminal, navigate to your project folder, and run this command:

```
php artisan migrate:make create_feeds_table --table=feeds --create
```

This command will generate a new database migration named `feeds` for us. Now navigate to `app/database/migrations`, open the migration file just created by the preceding command, and change its contents as follows:

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

class CreateFeedsTable extends Migration {

  /**
   * Run the migrations.
   *
   * @return void
   */
  public function up()
  {
    Schema::create('feeds', function(Blueprint $table)
    {
      $table->increments('id');
      $table->enum('active', array('0', '1'));
      $table->string('title,100)->default('');
      $table->enum('category', array('News', 'Sports','Technology'));
      $table->string('feed',1000)->default('');
      $table->timestamps();
    });
  }

  /**
   * Reverse the migrations.
   *
   * @return void
   */
  public function down()
  {
    Schema::drop('feeds');
  }

}
```

We have a `title` column to show titles on the website, which is more user-friendly. Also, we set a key named `active` because we want to enable/disable feeds; we set it with the new `enum()` method of Laravel, which is featured with Laravel 4. We also set a `category` column that is also set with the `enum()` method to group feeds.

After saving the file, run the following command to execute migration:

```
php artisan migrate
```

If no error occurs, you are ready for the next step of the project.

Creating a feeds model

As you know, for anything related to database operations on Laravel, using models is the best practice. We will benefit from the Eloquent ORM.

Save this file as `feeds.php` under `app/models/`:

```
<?php
Class Feeds Extends Eloquent{
    protected $table = 'feeds';
    protected $fillable = array('feed', 'title', 'active','category');
}
```

We set the table name and the fillable columns with values. Now that our model is ready, we can proceed to the next step, and start creating our controller, along with the form.

Creating our form

Now we should create a form to save records to the database and specify its properties.

1. First, open your terminal and enter the following command:

   ```
   php artisan controller:make FeedsController
   ```

   This command will generate a `FeedsController.php` file for you with some blank methods in the `app/controllers` folder.

The default methods in the controller that are auto-filled by the `artisan` commands are not RESTful.

2. Now, open `app/routes.php` and add the following lines:

   ```
   //We defined a RESTful controller and all its via route directly
   Route::controller('feeds', 'FeedsController');
   ```

Instead of defining all actions one by one, we can define all actions declared on a controller with a line of code. If your method names are usable as get or post actions directly, using the `controller()` method can save you a large amount of time. The first parameter sets the URI for the controller and the second parameter defines which class in the `controllers` folder will be accessed and defined.

Controllers that are set in this manner are automatically RESTful.

3. Now, let's create the form's method. Add these lines of code into your controller file:

```
//The method to show the form to add a new feed
public function getCreate() {
  //We load a view directly and return it to be served
  return View::make('create_feed');
    }
```

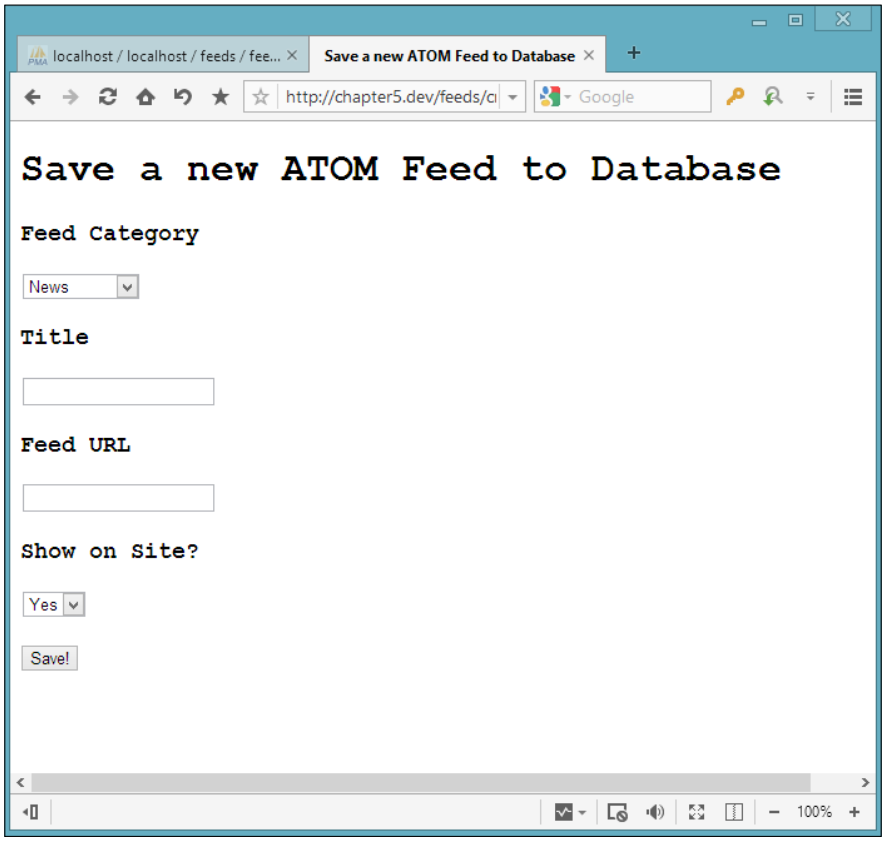
The process is quite simple here; we named the method as `getCreate()` because we want our `create` method to be RESTful. We simply loaded a view file, which we will be generating in the next step directly.

4. Now let's create our view file. Save this file as `create_feed.blade.php` under `app/views/`:

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Save a new ATOM Feed to Database</title>
</head>
<body>
  <h1>Save a new ATOM Feed to Database</h1>
  @if(Session::has('message'))
    <h2>{{Session::get('message')}}</h2>
  @endif
    {{Form::open(array('url' => 'feeds/create', 'method' =>
'post'))}}
    <h3>Feed Category</h3>
  {{Form::select('category',array('News'=>'News','Sports'=>'Sports
','Technology'=>'Technology'),Input::old('category'))}}
  <h3>Title</h3>
    {{Form::text('title',Input::old('title'))}}
```

```
    <h3>Feed URL</h3>
    {{Form::text('feed',Input::old('feed'))}}


    <h3>Show on Site?</h3>
{{Form::select('active',array('1'=>'Yes','2'=>'No'),Input::old('ac
tive'))}}
    {{Form::submit('Save!',array('style'=>'margin:20px 100% 0
0'))}}
    {{Form::close()}}
</body>
</html>
```

The preceding code will produce a simple form, shown as follows:



Validating and processing the form

In this section, we will validate the submitted form and ensure that the fields are valid and the required fields are filled. Then we will save the data to the database.

1. First, we need to define the form validation rules. We prefer adding validation rules to the related model, so the rules become reusable, and this prevents the code from becoming bloated. To do this, add the following code in `feeds.php` located at `app/models/` (the model that we generated earlier in this chapter), inside the class definition before the last `}`:

```
//Validation rules
public static $form_rules = array(
  'feed'    => 'required|url|active_url',
  'title'  => 'required'
  'active'  => 'required|between:0,1',
  'category'  => 'required| in:News,Sports,Technology'
);
```

   We set the variable as `public`, so it can be used outside the model's file itself, and we set it to `static`, so we can directly access the variable.

   We want the feed to be a URL, and we want to check whether it's an active URL or not using the `active_url` validation rule, which depends on PHP's `chkdnsrr()` method.

   Our active field can only get two values, `1` or `0`. Since we set it with integers, we can use the `between` rule of Laravel's form validation and check whether the number is between `1` and `0`.

   Our category field also has the `enum` type, and its value should only be `News`, `Sports`, or `Technology`. To check the exact values with Laravel, you can use the validation rule `in`.

Not all server configurations support the `chkdnsrr()` method, so make sure it's installed on your side, else you may depend on only validating it if the URL is formatted correctly.

2. Now we need a controller post method to process the form. Add the following method to `app/controllers/FeedsController.php` before the last `}`:

```
//Processing the form
public function postCreate(){

//Let's first run the validation with all provided input
  $validation = Validator::make(Input::all(),Feeds::$form_rules);
```

```
  //If the validation passes, we add the values to the database and
return to the form
  if($validation->passes()) {
    //We try to insert a new row with Eloquent
    $create = Feeds::create(array(
      'feed'    => Input::get('feed'),
      'title'  => Input::get('title'),
      'active'  => Input::get('active'),
      'category'  => Input::get('category')
    ));

    //We return to the form with success or error message due to state
of the
    if($create) {
      return Redirect::to('feeds/create')
        ->with('message','The feed added to the database
successfully!');
    } else {
      return Redirect::to('feeds/create')
        ->withInput()
        ->with('message','The feed could not be added, please try
again later!');
    }
  } else {
    //If the validation does not pass, we return to the form with
first error message as flash data
    return Redirect::to('feeds/create')
        ->withInput()
        ->with('message',$validation->errors()->first());

  }
}
```

Let's dig into the code one by one. First, we made a form validation and called our validation rules from the model that we've generated via `Feeds::$form_rules`.

After that, we created an `if()` statement and divided the code into two parts with it. If the form validation fails, we return to the form with old inputs using the `withInput()` special method, and add a flash data message field using the method `with()`.

If the form validation passes, we try to add a new column to the database with Eloquent's `create()` method, and we return to the form with a success or error message depending on what the `create` method returns.

Now, we need to make a new view for the index page, which will show the last five entries of all the feeds. But before that, we need a function to parse the Atom feeds. For this, we will be extending the build in the `Str` class of Laravel.

Extending the core classes

Laravel has many built-in methods that make our life easier. But as in all bundled packages, the bundle itself may not satisfy any of its users as it is introduced. So, you may want to use your own methods along with the bundled ones. You can always create new classes, but what if half of what you want to achieve is already built-in? For example, you want to add a form element, but there is already a `Form` class bundled. In this case, you may want to extend the current class(es) with your own methods instead of creating new ones to keep the code tidy.

In this section, we will be extending the `Str` class with the method called `parse_atom()`, which we will code.

1. First, you must find where the class file is. We will be extending the `Str` class, which is under `vendor/laravel/framework/src/Illuminate/Support`. Note that you can also find this class under the aliases key in `app/config/app.php`.
2. Now create a new folder called `lib` under `app/folder`. This folder will hold our class extensions. Because the `Str` class is grouped under the folder `Support`, it is suggested that you create a new folder named `Support` under `lib` too.
3. Now create a new file named `Str.php` under `app/lib/Support`, which you've just created:

   ```
   <?php namespace app\lib\Support;
   class Str extends \Illuminate\Support\Str {
       //Our shiny extended codes will come here
     }
   ```

   We gave a namespace to it so we can access it easily. Instead of using it like `\app\lib\Support\Str::trim()` (which you can), you can directly use it like `Str::trim()`. The rest of the code explains how to extend the library. We have provided the class name starting from the `Illuminate` path to access the `Str` class directly.

4. Now open your `app.php` file located at `app/config/`; comment out the following line:

   ```
   'Str'             => 'Illuminate\Support\Str',
   ```

5. Now, add the following line:

```
'Str'             => 'app\lib\Support\Str',
```

   This way, we switched the autoloaded `Str` class with our class, which is already extending the original.

6. Now to make it identifiable on autoruns, open your `composer.json` file and add these lines into autoload's `classmap` object:

```
"app/lib",
"app/lib/Support"
```

7. Finally, run the following command in the terminal:

```
php composer.phar dump-autoload
```

   This will look for dependencies and recompile common classes. If everything goes smoothly, you will now have an extended `Str` class.

Folder and class names are case-sensitive, even on Windows servers.

Reading and parsing an external feed

We have the feed URLs and the titles on our server all categorized. Now all we have to do is to parse them and show them to the end user. There are some steps to follow for this:

1. First, we need a method to parse external Atom feeds. Open your `Str.php` file located at `app/lib/Support/` and add this method into the class:

```
public static function parse_feed($url) {
    //First, we get our well-formatted external feed
    $feed = simplexml_load_file($url);
    //if cannot be found, or a parse/syntax error occurs, we
return a blank array
    if(!count($feed)) {
      return array();
    } else {
      //If found, we return the newest five <item>s in the
<channel>
      $out = array();
      $items = $feed->channel->item;
      for($i=0;$i<5;$i++) {
        $out[] = $items[$i];
```

```
        }
        //and we return the output
        return $out;
    }
}
```

First, we load the XML feed on the method using SimpleXML's built-in method, `simplexml_load_file()`. If no results are found or the feed contains errors, we return an empty array. In SimpleXML, all objects and their child objects are exactly like XML tags. So if there is a `<channel>` tag, there will be an object named `channel`, and if there are `<item>` tags inside `<channel>`, there will be an object named `item` beneath each `channel` object. So if you want to access the first item inside the channel, you can access it as `$xml->channel->item[0]`.

2. Now we need a view to show the contents. First, open your `routes.php` under `app` and delete the `get` route that is present by default:

```
Route::get('/', array('as'=>'index', 'uses' =>
'FeedsController@getIndex'));
```

3. Now open `FeedsController.php` located at `app/controller/` and paste this code:

```
public function getIndex(){
  //First we get all the records that are active category by
category:
    $news_raw   = Feeds::whereActive(1)->whereCategory('News')-
>get();
    $sports_raw  = Feeds::whereActive(1)->whereCategory('Sports')-
>get();
    $technology_raw = Feeds::whereActive(1)-
>whereCategory('Technology')->get();


  //Now we load our view file and send variables to the view
  return View::make('index')
    ->with('news',$news_raw)
    ->with('sports',$sports_raw)
    ->with('technology',$technology_raw);
  }
```

In the controller, we got the feeds' URLs one by one, and then loaded a view and set them one by one as separated variables for each of the categories.

4. Now we need to loop each feed category and show its contents. Save the following code in a file called `index.blade.php` under `app/views/`:

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Your awesome news aggregation site</title>
    <style type="text/css">
    body { font-family: Tahoma, Arial, sans-serif; }
    h1, h2, h3, strong { color: #666; }
    blockquote{ background: #bbb; border-radius: 3px; }
    li { border: 2px solid #ccc; border-radius: 5px; list-style-
type: none; margin-bottom: 10px }
    a { color: #1B9BE0; }
    </style>
</head>
<body>
   <h1>Your awesome news aggregation site</h1>
   <h2>Latest News</h2>
    @if(count($news))
        {{--We loop all news feed items --}}
        @foreach($news as $each)
            <h3>News from {{$each->title}}:</h3>
            <ul>
            {{-- for each feed item, we get and parse its feed
elements --}}
            <?php $feeds = Str::parse_feed($each->feed); ?>
            @if(count($feeds))
                {{-- In a loop, we show all feed elements one by
one --}}
                @foreach($feeds as $eachfeed)
                    <li>
                        <strong>{{$eachfeed->title}}</strong><br
/>
                        <blockquote>{{Str::limit(strip_
tags($eachfeed->description),250)}}</blockquote>
                        <strong>Date: {{$eachfeed->pubDate}}</
strong><br />
                        <strong>Source: {{HTML::link($eachfeed-
>link,Str::limit($eachfeed->link,35))}}</strong>

                    </li>
```

```
                @endforeach
            @else
                <li>No news found for {{$each->title}}.</li>
            @endif
            </ul>
        @endforeach
    @else
        <p>No News found :(</p>
    @endif

    <hr />

    <h2>Latest Sports News</h2>
    @if(count($sports))
        {{--We loop all news feed items --}}
        @foreach($sports as $each)
            <h3>Sports News from {{$each->title}}:</h3>
            <ul>
            {{-- for each feed item, we get and parse its feed
elements --}}
            <?php $feeds = Str::parse_feed($each->feed); ?>
            @if(count($feeds))
                {{-- In a loop, we show all feed elements one by
one --}}
                @foreach($feeds as $eachfeed)
                    <li>
                        <strong>{{$eachfeed->title}}</strong><br
/>
                        <blockquote>{{Str::limit(strip_
tags($eachfeed->description),250)}}</blockquote>
                        <strong>Date: {{$eachfeed->pubDate}}</
strong><br />
                        <strong>Source: {{HTML::link($eachfeed-
>link,Str::limit($eachfeed->link,35))}}</strong>
                    </li>
                @endforeach
            @else
                <li>No Sports News found for {{$each->title}}.</
li>
            @endif
            </ul>
        @endforeach
```

```
    @else
        <p>No Sports News found :(</p>
    @endif


    <hr />

    <h2>Latest Technology News</h2>
    @if(count($technology))
       {{--We loop all news feed items --}}
        @foreach($technology as $each)
            <h3>Technology News from {{$each->title}}:</h3>
            <ul>
            {{-- for each feed item, we get and parse its feed
elements --}}
            <?php $feeds = Str::parse_feed($each->feed); ?>
            @if(count($feeds))
                {{-- In a loop, we show all feed elements one by
one --}}
                @foreach($feeds as $eachfeed)
                    <li>
                        <strong>{{$eachfeed->title}}</strong><br
/>
                        <blockquote>{{Str::limit(strip_
tags($eachfeed->description),250)}}</blockquote>
                        <strong>Date: {{$eachfeed->pubDate}}</
strong><br />
                        <strong>Source: {{HTML::link($eachfeed-
>link,Str::limit($eachfeed->link,35))}}</strong>
                    </li>
                @endforeach
            @else
                <li>No Technology News found for {{$each-
>title}}.</li>
            @endif
            </ul>
        @endforeach
    @else
        <p>No Technology News found :(</p>
    @endif


</body>
</html>
```

5. We wrote the same code for each of the categories thrice. Also, between the `head` tags, a bit of styling is done, so the page will look prettier for the end user.

   We have divided each category's section with an `<hr>` tag. All three parts are working with the same mechanics of each other, except for the source variables and grouping.

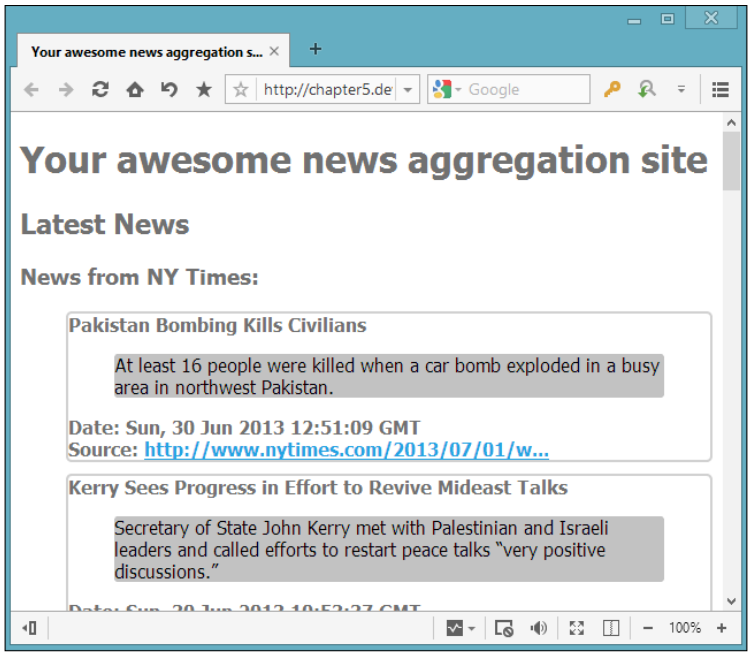
   We first checked whether records exist for each category (the results from the database, since we may not have added any news feeds yet). If there are results, they are looped through each record using the `@foreach()` method of Blade template engine.

   For each record, we first show the feed's friendly name (which we defined before while saving them) and parse the feed with the `parse_feed()` method we've just created .

   After we parse each feed, we look to see whether any records are found; if so, we loop them all again. To keep the tidiness of our feed reader, we trimmed all HTML tags with PHP's `strip_tags()` function and limited them to a maximum of 250 characters using the `limit()` method of Laravel's `Str` class (which we have extended).

   Individual feeds' items also have their own title, date, and source link, so we have displayed them as well on the feed. To prevent the link from breaking our interface, we limited the text, to be written between anchor tags, to 35 characters.

After all of the edits, you should get an output like this:



Summary

In this chapter, we've created a simple feeds reader using Laravel's built-in functions and PHP's `SimpleXML` class. We've learned how to extend core libraries, write our own methods, and use them in production. We also learned how to filter results while querying the database and how to create records. Finally, we learned how to work with strings, limit them, and clean them up. In the next chapter, we will be creating a photo gallery system. We will ensure that the uploaded files are photos. We will also group the photos into albums, and will relate albums and photos with Laravel's built-in relation methods.

6
Creating a Photo Gallery System

In this chapter, we'll code a simple photo gallery system with Laravel. We'll also cover Laravel's built-in file validation, file upload, and the **hasMany** database relation mechanism. We will use the `validation` class to validate the data and files. Also, we'll cover the file class for processing files. The following topics are covered in this chapter:

- Creating an Album model
- Creating an Image model
- Creating an album
- Creating a photo upload form
- Moving photos between albums

Creating a table and migrating albums

We assume that you've already defined the database credentials in the `database.php` file located at `app/config/`. To build a photo gallery system, we need a database that has two tables: `albums` and `images`. To create a new database, simply run the following SQL command:

```
CREATE DATABASE laravel_photogallery
```

After successfully creating the database for the application, we will first need to create the `albums` table and install it in the database. To do this, open up your terminal, navigate through your project folder, and run the following command:

```
php artisan migrate:make create_albums_table --table=albums --create
```

The preceding command will generate a migration file under `app/database/migrations` to generate a new MySQL table, named `posts`, in our `laravel_photogallery` database.

To define our table columns, we need to edit the migration file. After editing, the file should have the following code:

```
<?php

use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateAlbumsTable extends Migration {

  /**
  * Run the migrations.
  *
  * @return void
  */
  public function up()
    {
      Schema::create('albums', function(Blueprint $table)
      {
        $table->increments('id')->unsigned();
        $table->string('name');
        $table->text('description');
        $table->string('cover_image');
        $table->timestamps();
      });
    }

  /**
  * Reverse the migrations.
  *
  * @return void
  */
  public function down()
  {
    Schema::drop('albums');
  }
}
```

After saving the file, we need to use a simple artisan command again to execute migrations:

```
php artisan migrate
```

If no error has occurred, please check the `laravel_photogallery` database for the `albums` table and its columns.

Let's examine the columns in the following list:

- `id`: This column is used for storing the ID of the album
- `name`: This column is used for storing the name of the album
- `description`: This column is used for storing the description of the album
- `cover_image`: This column is for storing the cover image of the album

We've successfully created our `albums` table, so we need to code our **Album** model.

Creating an Album model

As you know, for anything related to database operations on Laravel, using models is the best practice. We will benefit from using the Eloquent ORM.

Save the following code as `Album.php` in the `app/models/` directory:

```
<?php
class Album extends Eloquent {

  protected $table = 'albums';

  protected $fillable = array('name','description','cover_image');

  public function Photos(){

    return $this->has_many('images');
  }
}
```

We have set the database table name using the `protected $table` variable; we've also set the editable columns using the `protected $fillable` variable, which we've already seen and used in previous chapters. The variables that are defined in the model are enough for using Laravel's Eloquent ORM. We'll cover the `public Photos ()` function in the *Assigning a photo to an album* section of this chapter.

Our **Album** model is ready; now we need an **Image** model and a database to assign photos to albums. Let's create them.

Creating the images database with the migrating class

To create our migration file for images, open up your terminal, navigate through your project folder, and run the following command:

```
php artisan migrate:make create_images_table --table=images --create
```

As you know, the command will generate a migration file in `app/database/migrations`. Let's edit the migration file; the final code should be as follows:

```
<?php

use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateImagesTable extends Migration {

  /**
  * Run the migrations.
  *
  * @return void
  */
  public function up()
  {
    Schema::create('images', function(Blueprint $table)
    {
      $table->increments('id')->unsigned();
      $table->integer('album_id')->unsigned();
      $table->string('image');
      $table->string('description');
      $table->foreign('album_id')->references('id')
        ->on('albums')->onDelete('CASCADE')
        ->onUpdate('CASCADE');
      $table->timestamps();
    });
  }

  /**
  * Reverse the migrations.
  *
```

```
     * @return void
     */
     public function down()
     {
       Schema::drop('images');
     }
   }
```

After editing the migration file, run the following migrate command:

```
php artisan migrate
```

As you know, the command creates the `images` table and its columns. If no error has occurred, check the `laravel_photogallery` database for the `users` table and the columns.

Let's examine the columns in the following list:

- `id`: This column is used for storing the id of the image
- `album_id`: This column is used for storing the id of the image's album
- `description`: This column is used for storing the description of the image
- `image`: This column is used for storing the path of the image

We need to explain one more thing for this migration file. As you can see in the migration code, there is a `foreign` key. We use the `foreign` key when we need to link two tables. We have an `albums` table and each album will have images. If the album is deleted from the database, you want all its images to be deleted as well.

Creating an Image model

We've already created the `images` table. So, as you know, we need a model to operate database tables on Laravel. To create that, save the following code as Image.php in the `app/models/` directory:

```
class Images extends Eloquent {

  protected $table = 'images';

  protected $fillable = array('album_id','description','image');

}
```

Our **Image** model is ready; now we need a controller to create the albums on our database. Let's create that.

Creating an album

As you know from the previous chapters in this book, Laravel has a great RESTful controller mechanism. We'll continue to use that to keep the code simple and short during development. In the next chapters, we'll cover another great controller/routing method named **Resource Controllers**.

To list, create, and delete an album, we need some functions in our controller. To create them, save the following code as `AlbumsController.php` in the `app/controllers/` directory:

```
<?php

class AlbumsController extends BaseController{

  public function getList()
  {
    $albums = Album::with('Photos')->get();
    return View::make('index')
    ->with('albums',$albums);
  }
  public function getAlbum($id)
  {
    $album = Album::with('Photos')->find($id);
    return View::make('album')
    ->with('album',$album);
  }
  public function getForm()
  {
    return View::make('createalbum');
  }
  public function postCreate()
  {
    $rules = array(

      'name' => 'required',
      'cover_image'=>'required|image'

    );

    $validator = Validator::make(Input::all(), $rules);
    if($validator->fails()){
```

```
        return Redirect::route('create_album_form')
        ->withErrors($validator)
        ->withInput();
      }

      $file = Input::file('cover_image');
      $random_name = str_random(8);
      $destinationPath = 'albums/';
      $extension = $file->getClientOriginalExtension();
      $filename=$random_name.'_cover.'.$extension;
      $uploadSuccess = Input::file('cover_image')
      ->move($destinationPath, $filename);
      $album = Album::create(array(
        'name' => Input::get('name'),
        'description' => Input::get('description'),
        'cover_image' => $filename,
      ));

      return Redirect::route('show_album',array('id'=>$album->id));
    }

    public function getDelete($id)
    {
      $album = Album::find($id);

      $album->delete();

      return Redirect::route('index');
    }
}
```

The `postCreate()` function first validates the posted data of the form. We'll cover validation next. If the data is validated successfully, we will rename the cover image and upload it with a new filename, because the code overwrites files which have the same name.

The `getDelete()` function is deleting the album along with assigned images (which are stored in an `images` table) from the database. Please remember the following migration file code:

```
$table->foreign('album_id')->references('id')->on('albums')
  ->onDelete('CASCADE')->onUpdate('CASCADE');
```

Before creating our templates, we need to define the routes. So, open up the `routes.php` file in the `app` folder and replace the code with the following one:

```
<?php
Route::get('/', array('as' => 'index',
  'uses' => 'AlbumsController@getList'));
Route::get('/createalbum', array('as' => 'create_album_form',
  'uses' => 'AlbumsController@getForm'));
Route::post('/createalbum', array('as' => 'create_album',
  'uses' => 'AlbumsController@postCreate'));
Route::get('/deletealbum/{id}', array('as' => 'delete_album',
  'uses' => 'AlbumsController@getDelete'));
Route::get('/album/{id}', array('as' => 'show_album',
  'uses' => 'AlbumsController@getAlbum'));
```

Now, we need some template files to show, create, and list the albums. First, we should create the index template. To create that, save the following code as `index.blade.php` in the `app/views/` directory:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Awesome Albums</title>
    <!-- Latest compiled and minified CSS -->
    <link href="//netdna.bootstrapcdn.com/bootstrap/
      3.0.0-rc1/css/bootstrap.min.css" rel="stylesheet">

    <!-- Latest compiled and minified JavaScript -->
    <script src="//netdna.bootstrapcdn.com/bootstrap/
      3.0.0-rc1/js/bootstrap.min.js"></script>
    <style>
      body {
        padding-top: 50px;
      }
      .starter-template {
        padding: 40px 15px;
      text-align: center;
      }
    </style>
  </head>
  <body>
    <div class="navbar navbar-inverse navbar-fixed-top">
      <div class="container">
      <button type="button" class="navbar-toggle"
        data-toggle="collapse" data-target=".nav-collapse">
```

```
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
        </button>
        <a class="navbar-brand" href="/">Awesome Albums</a>
        <div class="nav-collapse collapse">
          <ul class="nav navbar-nav">
            <li><a href="{{URL::route('create_album_form')}}">
              Create New Album</a></li>
          </ul>
        </div><!--/.nav-collapse -->
      </div>
      </div>

        <div class="container">

          <div class="starter-template">

          <div class="row">
            @foreach($albums as $album)
              <div class="col-lg-3">
                <div class="thumbnail" style="min-height: 514px;">
                  <img alt="{{$album->name}}" src="/albums/{
                    {$album->cover_image}}">
                  <div class="caption">
                    <h3>{{$album->name}}</h3>
                    <p>{{$album->description}}</p>
                    <p>{{count($album->Photos)}} image(s).</p>
                    <p>Created date:  {{ date("d F Y",strtotime
                      ($album->created_at)) }} at {{
                      date("g:ha",strtotime($album
                        ->created_at)) }}</p>
                    <p><a href="{{URL::route('show_album', array
                      ('id'=>$album->id))}}" class=
                      "btn btn-big btn-default">Show Gallery</a></p>
                  </div>
                </div>
              </div>
            @endforeach
          </div>

        </div><!-- /.container -->
      </div>

    </body>
  </html>
```

Adding a template for creating albums

As you can see in the following code, we prefer to use Twitter's bootstrap **CSS** framework. This framework allows you to rapidly create useful, responsive, and multi-browser supported interfaces. Next, we need to create a template for creating albums. To create that, save the following code as `createalbum.blade.php` in the `app/views/` directory:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
      initial-scale=1.0">
    <title>Create an Album</title>
    <!-- Latest compiled and minified CSS -->
    <link href="//netdna.bootstrapcdn.com/bootstrap/
      3.0.0-rc1/css/bootstrap.min.css" rel="stylesheet">

    <!-- Latest compiled and minified JavaScript -->
    <script src="//netdna.bootstrapcdn.com/bootstrap/
      3.0.0-rc1/js/bootstrap.min.js"></script>
  </head>
  <body>
    <div class="navbar navbar-inverse navbar-fixed-top">
      <div class="container">
        <button type="button" class="navbar-toggle"
          data-toggle="collapse" data-target=".nav-collapse">
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          <span lclass="icon-bar"></span>
        </button>
        <a class="navbar-brand" href="/">Awesome Albums</a>
        <div class="nav-collapse collapse">
          <ul class="nav navbar-nav">
            <li class="active"><a
              href="{{URL::route('create_album_form')}}">Create
              New Album</a></li>
          </ul>
        </div><!--/.nav-collapse -->
      </div>
    </div>
    <div class="container" style="text-align: center;">
      <div class="span4" style="display: inline-block;
        margin-top:100px;">
```

```
@if($errors->has())
  <div class="alert alert-block alert-error fade in"
    id="error-block">
     <?php
     $messages = $errors->all('<li>:message</li>');
    ?>
    <button type="button" class="close"
      data-dismiss="alert">x</button>

    <h4>Warning!</h4>
    <ul>
      @foreach($messages as $message)
        {{$message}}
      @endforeach

    </ul>
  </div>
@endif

<form name="createnewalbum" method="POST"
  action="{{URL::route('create_album')}}"
  enctype="multipart/form-data">
  <fieldset>
    <legend>Create an Album</legend>
    <div class="form-group">
      <label for="name">Album Name</label>
      <input name="name" type="text" class="form-control"
        placeholder="Album Name"
        value="{{Input::old('name')}}">
    </div>
    <div class="form-group">
      <label for="description">Album Description</label>
      <textarea name="description" type="text"
        class="form-control" placeholder="Album
        description">{{Input::old('descrption')}}
        </textarea>
    </div>
    <div class="form-group">
      <label for="cover_image">Select a Cover Image
      </label>
      {{Form::file('cover_image')}}
    </div>
    <button type="submit" class="btn
      btn-default">Create!</button>
  </fieldset>
```

```
          </form>
        </div>
      </div> <!-- /container -->
    </body>
  </html>
```

The template creates a basic upload form and shows the validation errors which are passed from the controller side. We need just one more template file to list the album images. So, to create it, save the following code as `album.blade.php` in the `app/views/` directory:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>{{$album->name}}</title>
    <!-- Latest compiled and minified CSS -->
    <link href="//netdna.bootstrapcdn.com/bootstrap/
      3.0.0-rc1/css/bootstrap.min.css" rel="stylesheet">

    <!-- Latest compiled and minified JavaScript -->
    <script src="//netdna.bootstrapcdn.com/bootstrap/
      3.0.0-rc1/js/bootstrap.min.js"></script>
    <style>
      body {
        padding-top: 50px;
      }
      .starter-template {
        padding: 40px 15px;
        text-align: center;
      }
    </style>
  </head>
  <body>
    <div class="navbar navbar-inverse navbar-fixed-top">
      <div class="container">
        <button type="button" class="navbar-toggle"
          data-toggle="collapse" data-target=".nav-collapse">
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
        </button>
        <a class="navbar-brand" href="/">Awesome Albums</a>
        <div class="nav-collapse collapse">
```

```
        <ul class="nav navbar-nav">
          <li><a href="{{URL::route('create_album_form')}}">
            Create New Album</a></li>
        </ul>
      </div><!--/.nav-collapse -->
    </div>
  </div>
  <div class="container">

    <div class="starter-template">
      <div class="media">
        <img class="media-object pull-left"
          alt="{{$album->name}}" src="/albums/{{
          $album->cover_image}}" width="350px">
        <div class="media-body">
          <h2 class="media-heading" style="font-size: 26px;">
            Album Name:</h2>
          <p>{{$album->name}}</p>
        <div class="media">
        <h2 class="media-heading" style="font-size: 26px;">Album
          Description :</h2>
        <p>{{$album->description}}<p>
        <a href="{{URL::route('add_image',array('id'=>
          $album->id))}}"><button type="button"
          class="btn btn-primary btn-large">
          Add New Image to Album</button></a>
        <a href="{{URL::route('delete_album',array('id'=>
          $album->id))}}" onclick="return confirm('Are you
          sure?')"><button type="button"
          class="btn btn-danger btn-large">Delete Album
          </button></a>
      </div>
    </div>
  </div>
  </div>
  <div class="row">
    @foreach($album->Photos as $photo)
      <div class="col-lg-3">
        <div class="thumbnail" style="max-height: 350px;
          min-height: 350px;">
          <img alt="{{$album->name}}" src="/albums/{
            {$photo->image}}">
          <div class="caption">
            <p>{{$photo->description}}</p>
```

```
                <p><p>Created date:  {{ date("d F Y",strtotime(
                  $photo->created_at)) }} at {{ date
                  ("g:ha",strtotime($photo->created_at)) }}
                  </p></p>
                <a href="{{URL::route('delete_image',array('id'=>
                  $photo->id))}}" onclick="return confirm('Are you
                  sure?')"><button type="button" class="btn
                  btn-danger btn-small">Delete Image </button></a>
              </div>
            </div>
          </div>
        @endforeach
      </div>
    </div>

  </body>
</html>
```

As you may remember, we have used the `hasMany()` Eloquent method on our model side. On the controller side, we use the function as follows:

```
$albums = Album::with('Photos')->get();
```

The code fetches the whole image data in an array that belongs to the album. Because of that, we use the `foreach` loop in the following template:

```
@foreach($album->Photos as $photo)
  <div class="col-lg-3">
    <div class="thumbnail" style="max-height: 350px;
      min-height: 350px;">
    <img alt="{{$album->name}}" src="/albums/{
      {$photo->image}}">
      <div class="caption">
        <p>{{$photo->description}}</p>
        <p><p>Created date:  {{ date("d F Y",strtotime(
          $photo->created_at)) }} at {{ date
          ("g:ha",strtotime($photo->created_at)) }}</p></p>
        <a href="{{URL::route('delete_image',array('id'=>
          $photo->id))}}" onclick="return confirm('Are you
          sure?')"><button type="button" class="btn
          btn-danger btn-small">Delete Image</button></a>
      </div>
    </div>
  </div>
@endforeach
```

Creating a photo upload form

Now we need to create a photo upload form. We'll upload the photos and assign them to the albums. Let's first set the routes; open the `routes.php` file in the `app` folder and add the following code:

```
Route::get('/addimage/{id}', array('as' => 'add_image',
  'uses' => 'ImagesController@getForm'));
Route::post('/addimage', array('as' => 'add_image_to_album',
  'uses' => 'ImagesController@postAdd'));
Route::get('/deleteimage/{id}', array('as' => 'delete_image',
  'uses' => 'ImagesController@getDelete'));
```

We need a template for the photo upload form. To create that, save the following code as `addimage.blade.php` in the `app/views/` directory:

```
<!doctype html>
  <html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
      initial-scale=1.0">
    <title>Laravel PHP Framework</title>
    <!-- Latest compiled and minified CSS -->
    <link href="//netdna.bootstrapcdn.com/bootstrap/
      3.0.0-rc1/css/bootstrap.min.css" rel="stylesheet">

    <!-- Latest compiled and minified JavaScript -->
    <script src="//netdna.bootstrapcdn.com/bootstrap/
      3.0.0-rc1/js/bootstrap.min.js"></script>
  </head>
  <body>

    <div class="container" style="text-align: center;">
      <div class="span4" style="display: inline-block;
        margin-top:100px;">
        @if($errors->has())
          <div class="alert alert-block alert-error fade in"
            id="error-block">
            <?php
            $messages = $errors->all('<li>:message</li>');
            ?>
            <button type="button" class="close"
              data-dismiss="alert">x</button>
```

```
            <h4>Warning!</h4>
            <ul>
              @foreach($messages as $message)
                {{$message}}
              @endforeach

            </ul>
          </div>
        @endif
        <form name="addimagetoalbum" method="POST"
            action="{{URL::route('add_image_to_album')}}"
          enctype="multipart/form-data">
          <input type="hidden" name="album_id"
          value="{{$album->id}}" />
          <fieldset>
            <legend>Add an Image to {{$album->name}}</legend>
            <div class="form-group">
              <label for="description">Image Description</label>
              <textarea name="description" type="text"
                class="form-control" placeholder="Image
                description"></textarea>
            </div>
            <div class="form-group">
              <label for="image">Select an Image</label>
              {{Form::file('image')}}
            </div>
            <button type="submit" class="btn
              btn-default">Add Image!
              </button>
          </fieldset>
        </form>
      </div>
    </div> <!-- /container -->
  </body>
</html>
```

Before creating the template, we need to code our controller. So, save the following code as `ImageController.php` in the `app/controllers/` directory:

```
<?php
class ImagesController extends BaseController{

  public function getForm($id)
  {
    $album = Album::find($id);
    return View::make('addimage')
```

```
        ->with('album',$album);
    }

    public function postAdd()
    {
      $rules = array(

        'album_id' => 'required|numeric|exists:albums,id',
        'image'=>'required|image'

      );

      $validator = Validator::make(Input::all(), $rules);
      if($validator->fails()){

        return Redirect::route('add_image',array('id' =>
          Input::get('album_id')))
        ->withErrors($validator)
        ->withInput();
      }

      $file = Input::file('image');
      $random_name = str_random(8);
      $destinationPath = 'albums/';
      $extension = $file->getClientOriginalExtension();
      $filename=$random_name.'_album_image.'.$extension;
      $uploadSuccess = Input::file('image')
        ->move($destinationPath, $filename);
      Image::create(array(
        'description' => Input::get('description'),
        'image' => $filename,
        'album_id'=> Input::get('album_id')
      ));

      return Redirect::route('show_album',
        array('id'=>Input::get('album_id')));
    }
    public function getDelete($id)
    {
      $image = Image::find($id);
      $image->delete();
      return Redirect::route('show_album',
        array('id'=>$image->album_id));
    }
}
```

The controller has three functions; the first one is the `getForm()` function. This function basically shows our photo upload form. The second one validates and inserts the data into the database. We'll explain the validating and inserting functions in the next section. The third one is the `getDelete()` function. This function basically deletes the image records from the database.

Validating the photo

Laravel has a powerful validation library, which has been mentioned in this book many times. We validate the data in controllers as follows:

```
$rules = array(

  'album_id' => 'required|numeric|exists:albums,id',
  'image'=>'required|image'

);

$validator = Validator::make(Input::all(), $rules);
if($validator->fails()){

  return Redirect::route('add_image',array('id' =>
    Input::get('album_id')))
  ->withErrors($validator)
  ->withInput();
}
```

Let's examine the code. We defined some rules in `array`. We have two validation rules in the `rules` array. The first rule is as follows:

```
'album_id' => 'required|numeric|exists:albums,id'
```

The preceding rule means that the `album_id` field is required (must be posted in the form), it must be a numeric value, and must exist in the `id` column of the `albums` table as we want to assign images to `albums`. The second rule is as follows:

```
'image'=>'required|image'
```

The preceding rule means that the `image` field is required (must be posted in the form) and its content must be an image. Then we check the posted form data using the following code:

```
$validator = Validator::make(Input::all(), $rules);
```

The validation function needs two variables. The first one is the data that we need to validate. In this case, we set that using the `Input::all()` method, which means we need to validate the posted form data. The second one is the `rules` variable. The `rules` variable must be set as an array as shown in the following code:

```
$rules = array(

  'album_id' => 'required|numeric|exists:albums,id',
  'image'=>'required|image'

);
```

Laravel's validation class comes with many predefined rules. You can see the updated list of all available validation rules at `http://laravel.com/docs/validation#available-validation-rules`.

Sometimes, we need to validate only specific MIME types, such as `JPEG, BMP, ORG, and PNG`. You can easily set the validation rule for this kind of validation as shown in the following code:

```
'image' =>'required|mimes:jpeg,bmp,png'
```

Then we check the validation process using the following code:

```
if($validator->fails()){

  return Redirect::route('add_image',array('id' =>
    Input::get('album_id')))
  ->withErrors($validator)
  ->withInput();
}
```

If validation fails, we redirect the browser to the image upload form. Then, we show the rules in the template file using the following code:

```
@if($errors->has())
  <div class="alert alert-block alert-error fade in"
    id="error-block">
    <?php
    $messages = $errors->all('<li>:message</li>');
    ?>
    <button type="button" class="close"
      data-dismiss="alert">x</button>
```

```
        <h4>Warning!</h4>
        <ul>
          @foreach($messages as $message)
            {{$message}}
          @endforeach

        </ul>
      </div>
    @endif
```

Assigning a photo to an album

The `postAdd( )` function is used for processing the request to create a new image record in the database. We get the author's ID using the following previously mentioned method:

```
Auth::user()->id
```

Using the following method, we assign the current user with the blog post. We have a new method in the query as shown in the following code:

```
Posts::with('Author')->…
```

We've defined a `public Photos ()` function in our Album model using the following code:

```
public function Photos(){

  return $this->hasMany('images','album_id');
}
```

The `hasMany()` method is an Eloquent function for creating relations between tables. Basically, the function has one `required` variable and one optional variable. The first variable (`required`) is for defining the target model. The second, which is the optional variable, is for defining the source column of the current model's table. In this case, we store the albums' IDs in the `album_id` column in the `images` table. Because of that, we need to define the second variable as `album_id` in the function. The second parameter is only required if your ID doesn't follow the convention. Using this method, we can pass our albums' information and assigned images' data to the template at the same time.

As you can remember from *Chapter 4*, *Building a Personal Blog*, we can list the relational data in the `foreach` loop. Let's have a quick look at the image-listing section of code in our template file, which is located in `app/views/ album.blade.php`:

```
@foreach($album->Photos as $photo)

  <div class="col-lg-3">
    <div class="thumbnail" style="max-height: 350px;
      min-height: 350px;">
    <img alt="{{$album->name}}" src="/albums/{{$photo->image}}">
      <div class="caption">
        <p>{{$photo->description}}</p>
        <p><p>Created date:  {{ date("d F Y",
          strtotime($photo->created_at)) }} at {{ date
          ("g:ha",strtotime($photo->created_at)) }}</p></p>
        <a href="{{URL::route('delete_image',array('id'=>
          $photo->id))}}" onclick="return confirm('Are you
          sure?')"><button type="button" class="btn
          btn-danger btn-small">Delete Image</button></a>
      </div>
    </div>
  </div>

@endforeach
```

Moving photos between albums

Moving photos between albums is a great feature for managing the album's images. Many photo gallery systems come with this feature. So, we can code it easily with Laravel. We need a form and controller function for adding this feature to our photo gallery system. Let's code the controller function first. Open the `ImagesController.php` file which is located in `app/controllers/` and add the following code in it:

```
public function postMove()
{
  $rules = array(

    'new_album' => 'required|numeric|exists:albums,id',
    'photo'=>'required|numeric|exists:images,id'

  );
```

```
    $validator = Validator::make(Input::all(), $rules);
    if($validator->fails()){

      return Redirect::route('index');
    }
    $image = Image::find(Input::get('photo'));
    $image->album_id = Input::get('new_album');
    $image->save();
    return Redirect::route('show_album',
      array('id'=>Input::get('new_album')));
  }
```

As you can see in the preceding code, we use the `Validation` class again. Let's examine the rules. The first rule is as follows:

```
'new_album' => 'required|numeric|exists:albums,id'
```

The preceding rule means that the `new_album` field is `required` (must be posted in the form), must be a numeric value, and exist in the `id` column of the `albums` table. We want to assign images to albums, so the images must exist. The second rule is as follows:

```
'photo'=>'required|numeric|exists:images,id'
```

The preceding rule means that the `photo` field is `required` (must be posted in the form), must be a numeric value, and exist in the `id` column of the `images` table.

After successful validation, we update the `album_id` column of the `photos` field and redirect the browser to show the new album of photos using the following code:

```
$image = Image::find(Input::get('photo'));
$image->album_id = Input::get('new_album');
$image->save();
return Redirect::route('show_album',
  array('id'=>Input::get('new_album')));
```

The final code of the `Images` controller should be as follows:

```
<?php

class ImagesController extends BaseController{

  public function getForm($id)
  {
    $album = Album::find($id);

    return View::make('addimage')
```

```
    ->with('album',$album);
}

public function postAdd()
{
  $rules = array(

    'album_id' => 'required|numeric|exists:albums,id',
    'image'=>'required|image'

  );

  $validator = Validator::make(Input::all(), $rules);
  if($validator->fails()){

    return Redirect::route('add_image',array('id' =>
      Input::get('album_id')))
    ->withErrors($validator)
    ->withInput();
  }

  $file = Input::file('image');
  $random_name = str_random(8);
  $destinationPath = 'albums/';
  $extension = $file->getClientOriginalExtension();
  $filename=$random_name.'_album_image.'.$extension;
  $uploadSuccess = Input::file('image')->move(
    $destinationPath, $filename);
  Image::create(array(
    'description' => Input::get('description'),
    'image' => $filename,
    'album_id'=> Input::get('album_id')
  ));

  return Redirect::route('show_album',
    array('id'=>Input::get('album_id')));
}
public function getDelete($id)
{
  $image = Image::find($id);

  $image->delete();

  return Redirect::route('show_album',
    array('id'=>$image->album_id));
}
public function postMove()
```

```
    {
      $rules = array(
        'new_album' => 'required|numeric|exists:albums,id',
        'photo'=>'required|numeric|exists:images,id'
      );
      $validator = Validator::make(Input::all(), $rules);
      if($validator->fails()){

        return Redirect::route('index');
      }
      $image = Image::find(Input::get('photo'));
      $image->album_id = Input::get('new_album');
      $image->save();
      return Redirect::route('show_album',
        array('id'=>Input::get('new_album')));
    }
  }
```

Our controller is ready, so we need to set up the updated form's route in `app/routes.php`. Open the file and add the following code:

```
Route::post('/moveimage', array('as' => 'move_image', 'uses' =>
'ImagesController@postMove'));
```

The final code located in `app/routes.php` should look as follows:

```
<?php
Route::get('/', array('as' => 'index', 'uses' =>
  'AlbumsController@getList'));
Route::get('/createalbum', array('as' => 'create_album_form',
  'uses' => 'AlbumsController@getForm'));
Route::post('/createalbum', array('as' => 'create_album',
  'uses' => 'AlbumsController@postCreate'));
Route::get('/deletealbum/{id}', array('as' => 'delete_album',
  'uses' => 'AlbumsController@getDelete'));
Route::get('/album/{id}', array('as' => 'show_album', 'uses' =>
  'AlbumsController@getAlbum'));
Route::get('/addimage/{id}', array('as' => 'add_image', 'uses' =>
  'ImagesController@getForm'));
Route::post('/addimage', array('as' => 'add_image_to_album',
  'uses' => 'ImagesController@postAdd'));
Route::get('/deleteimage/{id}', array('as' => 'delete_image',
'uses' => 'ImagesController@getDelete'));
Route::post('/moveimage', array('as' => 'move_image',
'uses' => 'ImagesController@postMove'));
```

Creating an update form

Now we need to create our update form in our template file. Open the template file which is located in `app/views/ album.blade.php` and change the `foreach` loop as follows:

```
@foreach($album->Photos as $photo)
  <div class="col-lg-3">
    <div class="thumbnail" style="max-height: 350px;
      min-height: 350px;">
      <img alt="{{$album->name}}" src="/albums/{{$photo->image}}">
      <div class="caption">
        <p>{{$photo->description}}</p>
        <p>Created date:  {{ date("d F Y",
          strtotime($photo->created_at)) }}
          at {{ date("g:ha",strtotime($photo->created_at)) }}</p>
        <a href="{{URL::route('delete_image',
          array('id'=>$photo->id))}}" onclick="return
          confirm('Are you sure?')"><button type="button"
          class="btn btn-danger btn-small">Delete Image
          </button></a>
        <p>Move image to another Album :</p>
        <form name="movephoto" method="POST"
          action="{{URL::route('move_image')}}">
          <select name="new_album">
            @foreach($albums as $others)
              <option value="{{$others->id}}">{{$others->name}}
                </option>
            @endforeach
          </select>
          <input type="hidden" name="photo"
            value="{{$photo->id}}" />
          <button type="submit" class="btn btn-small
            btn-info" onclick="return confirm('Are you sure?')">
            Move Image</button>
        </form>
      </div>
    </div>
  </div>
@endforeach
```

Summary

In this chapter, we've created a simple photo gallery system with Laravel's built-in functions and the Eloquent database driver. We've learned how to validate the data, and about the powerful data relation method in Eloquent named hasMany. In the next chapters, we'll learn to work with more complex tables, and relational data and relation types.

7
Creating a Newsletter System

In this chapter, we will cover an advanced newsletter system, which will use Laravel's `queue` and `email` libraries. After this section, we will learn how to set and fire/trigger queued tasks, and how to parse e-mail templates and send mass e-mails to subscribers. The topics covered in this chapter are:

- Creating a database and migrating the subscriber's table
- Creating a subscriber's model
- Creating our subscription form
- Validating and processing the form
- Creating a queue system to process the e-mail
- Using the Email class to process e-mails inside the queue
- Testing the system
- Sending e-mails with the queue directly

In this chapter, we will be using third-party services, which will require access to your script, so before proceeding, make sure your project is available online.

Creating a database and migrating the subscribers table

After successfully installing Laravel 4 and defining database credentials from `app/config/database.php`, create a database called `chapter7`.

After creating the database, open up your terminal and navigate through your project folder, and run the following command:

```
php artisan migrate:make create_subscribers_table --table=subscribers
--create
```

The preceding command will generate a new MySQL migration named `subscribers` for us. Now navigate to the `migrations` folder in `app/database/` and open up the migration file just created by the preceding command, and change its contents as shown in the following code:

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

class CreateSubscribersTable extends Migration {

  /**
   * Run the migrations.
   *
   * @return void
   */
  public function up()
  {
    Schema::create('subscribers', function(Blueprint $table)
    {
      $table->increments('id');
      $table->string('email,100)->default('');
      $table->timestamps();
    });
  }

  /**
  * Reverse the migrations.
  *
  * @return void
  */
  public function down()
  {
    Schema::drop('subscribers');
  }
}
```

For this chapter, we will only need the `email` column, which will hold the e-mail address, of the subscribers. I set this column to be, at most, 100 characters long, having datatype `VARCHAR`, and it is not to be left as null.

After saving the file, run the following command to execute migration:

```
php artisan migrate
```

If no error has occurred, you are ready for the next step of the project.

Creating a subscribers model

To benefit from Eloquent ORM, the best practice is to create a model.

Save the following code in `subscribers.php` at `app/models/`:

```
<?php
Class Subscribers Extends Eloquent{
  protected $table = 'subscribers';
  protected $fillable = array('email');
}
```

We set the table name with the variable `$table`, and columns in which the value must be filled by the user are set with the `$fillable` variable. Now that our model is ready, we can proceed to the next step, and start creating our controller, along with the form.

Creating our subscription form

Now we should create a form to save records to the database and specify its properties.

1. First, open your terminal and type the following command:

   ```
   php artisan controller:make SubscribersController
   ```

   This command will generate a `SubscribersController.php` file for you with some blank methods in the `app/controllers` directory.

The default controller methods generated by the `artisan` command are not RESTful.

2. Now, open up `app/routes.php` and add the following code:

   ```
   //We define a RESTful controller and all its via route
     //directly
   Route::controller('subscribers', 'SubscribersController');
   ```

   Instead of defining all actions one by one, we can define all actions declared on a controller with one line of code. If your method names are usable as `get` or `post` actions directly, using the `controller()` method can save a lot of time. The first parameter sets the **URI** (**Uniform Resource Identifier**) for the controller and the second parameter defines which class in the controllers folder will be accessed and defined.

Controllers which are set like this are automatically RESTful.

3. Now, let's create the form's controller. Remove all methods inside the auto-generated class and add the following code in your controller file:

```
//The method to show the form to add a new feed
public function getIndex() {
  //We load a view directly and return it to be served
  return View::make('subscribe_form');
}
```

First, we defined the process. It is quite simple here; we named the method as `getCreate()`, because we want our `Create` method to be RESTful. We simply loaded a view file, which we will be generating in the next step directly.

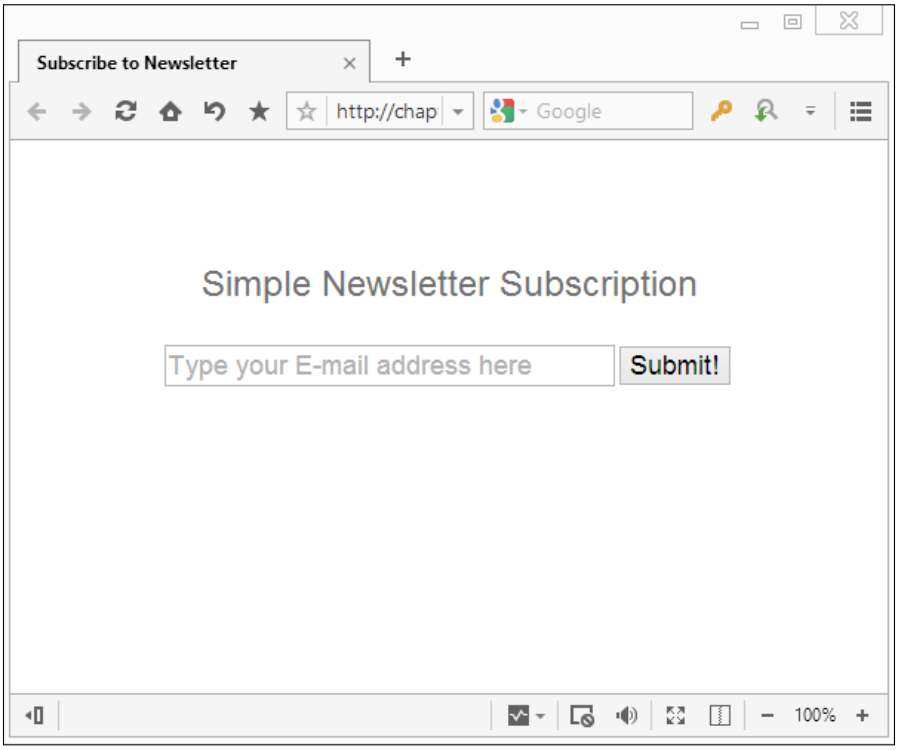
4. Now let's create our view file. In this example, I've used the Ajax POST technique using jQuery. Save this file as `subscribe_form.blade.php` at `app/views/`:

```
<!doctype html>
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Subscribe to Newsletter</title>
    <style>
      /*Some Little Minor CSS to tidy up the form*/
      body{margin:0;font-family:Arial,Tahoma,sans-serif;
        text-align:center;padding-top:60px;color:#666;
        font-size:24px}
      input{font-size:18px}
      input[type=text]{width:300px}
      div.content{padding-top:24px;font-weight:700;
        font-size:24px}
      .success{color:#0b0}
      .error{color:#b00}
    </style>
  </head>
  <body>

    {{-- Form Starts Here --}}
    {{Form::open(array('url'=> URL::to
      ('subscribers/submit'),'method' => 'post'))}}
```

```
<p>Simple Newsletter Subscription</p>
{{Form::text('email',null,array
  ('placeholder'=>'Type your E-mail address here'))}}
{{Form::submit('Submit!')}}

{{Form::close()}}
{{-- Form Ends Here --}}

{{-- This div will show the ajax response --}}
<div class="content"></div>
{{-- Because it'll be sent over AJAX, We add the
  jQuery source --}}
{{ HTML::script
  ('http://code.jquery.com/jquery-1.8.3.min.js') }}
<script type="text/javascript">
  //Even though it's on footer, I just like to make
    //sure that DOM is ready
  $(function(){
    //We hide de the result div on start
    $('div.content').hide();
    //This part is more jQuery Related. In short, we
      //make an Ajax post request and get the response
      //back from server
    $('input[type="submit"]').click(function(e){
      e.preventDefault();
      $.post('/subscribers/submit', {
        email: $('input[name="email"]').val()
      }, function($data){
        if($data=='1') {
          $('div.content').hide().removeClass
            ('success error').addClass('success').html
            ('You\'ve successfully subscribed to our
            newsletter').fadeIn('fast');
        } else {
          //This part echos our form validation errors
          $('div.content').hide().removeClass
            ('success error').addClass('error').html
            ('There has been an error occurred:<br />
            <br />'+$data).fadeIn('fast');
        }
      });
    });
    //We prevented to submit by pressing enter or any
      other way
    $('form').submit(function(e){
      e.preventDefault();
```

```
            $('input[type="submit"]').click();
          });
        });
      </script>
    </body>
</html>
```

The preceding code will produce a simple form as shown in the following screenshot:



Now that our form is ready, we can continue and process the form.

Validating and processing the form

Now that we have the form, we need to validate and store the data. We also need to check whether the request is an Ajax request. Also, we need to return successful code or error messages back to the form with Ajax methods, so that the end-user can understand what has happened at the backend.

Save the data inside `SubscribersController.php` at `app/controllers/`:

```
//This method is to process the form
public function postSubmit() {

  //we check if it's really an AJAX request
  if(Request::ajax()) {

    $validation = Validator::make(Input::all(), array(
      //email field should be required, should be in an email
        //format, and should be unique
      'email' => 'required|email|unique:subscribers,email'
    )
    );

    if($validation->fails()) {
      return $validation->errors()->first();
    } else {

      $create = Subscribers::create(array(
        'email' => Input::get('email')
      ));

      //If successful, we will be returning the '1' so the form
        //understands it's successful
      //or if we encountered an unsuccessful creation attempt,
        return its info
      return $create?'1':'We could not save your address to our
        system, please try again later';
    }

  } else {
    return Redirect::to('subscribers');
  }
}
```

The following points explain the preceding code:

1. With the `ajax()` method of the `Request` class, you can check whether the request is an Ajax request or not. If it's not an Ajax request, we are redirected back to our subscriber's page (the form itself).
2. If it's a valid request, then we run our form using the `make()` method of the `Validation` class. In this example, I've written the rules directly, but the best practice is to set them in models and call them to the controller directly. The rule `required` checks whether the field is filled. The rule `email` checks if the input is in a valid e-mail format, and lastly, the rule `unique` helps us to know whether the value is already on a row or not.

3. If the form validation fails, we return the first error message directly. Returned content will be Ajax's response that will be echoed out into our form page. Since the error message is an auto-generated meaningful text message, it's safe to use it in our example directly. This message will show all errors from our validation. For example, it will echo out if the field is not a valid e-mail address, or if the e-mail has been submitted to the database already.
4. If the form validation passes, we try to add the e-mail to our database with the `create()` method of Laravel's Eloquent ORM.

Creating a queue system for basic e-mail sending

Queues, which are featured in Laravel 4, are one of the best features that come with the framework. Imagine you have a long process, such as resizing all images, sending mass e-mails, or mass database operations. When you process these, they will take time. So why should we wait? Instead we will put these processes in a queue. With Laravel v4, this is quite easy to manage. In this section, we are going to create a simple queue and loop through the e-mails and will try to send an e-mail to each subscriber using the following steps:

1. First, we need a queue driver for our project. This may be **Amazon SQS**, **Beanstalkd**, or **Iron IO**. I chose Iron IO because, currently, it's the only queue driver that supports push queues. Then we need to get the package from packagist. Add `"iron-io/iron_mq": "dev-master"` to the `require` key of `composer.json`. It should look like the following code:

```
"require": {
     "laravel/framework": "4.0.*",
     "iron-io/iron_mq": "dev-master"
},
```

2. Now you should run the following command to update/download new packages:

```
php composer.phar update
```

3. We need an account from one of the queue services that Laravel officially supports. In this example, I'll be using the free **Iron.io** service.
    1. First, sign up to the website `http://iron.io`.
    2. Second, after you're logged in, create a project named `laravel`.

3. Then, click on your project. There is a key icon that will give you the project's credentials. Click on that; it will provide you with `project_id` and `token`.

4. Now navigate to `app/config/queue.php`, and change the default key driver to iron.

   In the `queue` file that we opened, there is a key named `iron`, which you will be using to fill the credentials. Provide your `token` and `project_id` information there, and for the `queue` key, type `laravel`.

5. Now, open your terminal and type the following command:

   ```
   php artisan queue:subscribe laravel
     http://your-site-url/queue/push
   ```

6. If everything went okay, you will get an output as follows:

   ```
   Queue subscriber added: http://your-site-url/queue/push
   ```

7. Now, when you check the queues tab on the Iron.io project page, you will see a new `push` queue generated by Laravel. Because it's a push queue, the queue will call us when its time comes.

8. Now we need some methods to catch the `push` request, to marshal it, and to fire it.

   1. First, we will need a `get` method to trigger the `push` queue (which will mimic the codes to trigger the queue).

      Add the following code into your `routes.php` file in the `app` folder:

      ```
      //This code will trigger the push request
      Route::get('queue/process',function(){
        Queue::push('SendEmail');
        return 'Queue Processed Successfully!';
      });
      ```

      This code will make a `push` request to a class called `SendEmail`, which we will be creating in further steps.

   2. Now we will need a listener to marshal the queue. Add the following code into your `routes.php` file in the `app` folder:

      ```
      //When the push driver sends us back, we will have to
        //marshal and process the queue.
      Route::post('queue/push',function(){
        return Queue::marshal();
      });
      ```

This code will get the `push` request from our queue driver, which will put it up in the queue and run.

We will need a class to fire up the queue and send e-mails, but first we need an e-mail template. Save the code as `test.blade.php` in the `app/views/ emails/` directory:

```
<!DOCTYPE html>
<html lang="en-US">
  <head>
    <meta charset="utf-8">
  </head>
  <body>
    <h2>Welcome to our newsletter</h2>
    <div>Hello {{$email}}, this is our test message from
      our Awesome Laravel queue system.</div>
  /body>
</html>
```

This is a simple e-mail template that will wrap our e-mail.

3. Now we need a class to fire up the queue and send the e-mail. Save these class files directly into the `routes.php` file in the `app` folder:

```
//When the queue is pushed and waiting to be marshalled,
we should assign a Class to make the job done
Class SendEmail {

  public function fire($job,$data) {

    //We first get the all data from our subscribers
      //database
    $subscribers = Subscribers::all();

    foreach ($subscribers as $each) {

      //Now we send an email to each subscriber
      Mail::send('emails.test',
        array('email'=>$each->email), function($message){

        $message->from('us@oursite.com', 'Our Name');

        $message->to($each->email);

      });
    }

    $job->delete();
  }
}
```

The class `SendEmail`, which we have written in the preceding code, will cover the queue job that we will be assigning. The method `fire()` is Laravel's own method to process the queue event. So when the queue is marshaled, the code inside the method `fire()` will be run. We can also pass parameters to `job` as a second parameter while we are calling the `Queue::push()` method.

With the help of Eloquent ORM, we have obtained all the subscriber methods from the database using the `all()` method, then with a `foreach` loop, we looped through all the records.

After `job` is processed successfully, at the bottom, we use the `delete()` method so the job won't be started again on the next queue call.

Before digging into the code further, we must learn the basics of Laravel 4's new feature, **Email class**.

Using the Email class to process e-mails inside the queue

Before proceeding further, we need to make sure that our e-mail credentials are correct and we have set all the values correctly. Open the `mail.php` file in the `app/config/` directory, and fill the settings according to your configuration:

- The parameter driver sets which e-mail driver is to be used; `mail`, `sendmail`, and `smtp` are the default mail-sending parameters.
- If you are using `smtp`, you will need to fill the `host`, `port`, `encryption`, `username`, and `password` fields according to your provider.
- You can also set a default from-address with the field `from`, so you won't have to type the same address over and over again.
- If you're using `sendmail` as the mail-sending driver, you should make sure its path in the parameter `sendmail` is correct. Otherwise, the e-mails won't be sent.
- If you're still testing your application, or you are in a live environment and want to test your updates without the risk of sending the wrong/unfinished e-mails, you should set `pretend` to `true`, so instead of actually sending the e-mails, it will keep them on the logfiles for you to debug.

While we were looping through all the records, we used Laravel's new e-mail sender, the `Mail` class, which is based on the popular component, `Swiftmailer`.

The `Mail::send()` method takes three major parameters:

- The first parameter is the path of the e-mail template file in which the e-mail will be wrapped
- The second parameter is the variable that will be sent to the view
- The third parameter is a closure function, where we can set the titles `from`, `to`, `CC/BCC`, and `attachments`

Additionally, you can also use the method `attach()` to add attachments to the e-mail

Testing the system

After we set the queue system and the `email` class, we are ready to test the code we've written:

1. First, make sure there are some valid e-mail addresses in the database.
2. Now navigate through your browser and type `http://your-site-url/queue/process`.
3. When you see the message `Queue Processed`, this means the queue was sent to our queue driver successfully. I want to describe what's happening here, step by step:
    - First, we ping our queue driver containing `Queue::push()` with the parameters and additional data that we need to queue
    - Then, after the queue driver gets our response, it will make a post request to our post route `queue/push`, which we had set up with the `queue:subscribe` artisan command earlier
    - When the `push` request is received from the queue driver by our script, it marshals and triggers the queued event
    - After it's triggered, the method `fire()` that is inside the class runs and does the job that we assigned to it
4. After a while, if everything went okay, you will start to receive those e-mails in your inbox(es).

Sending e-mails with the queue directly

In some e-mail-sending cases, especially if we are using a third-party SMTP and if we are sending user registration, validation e-mails, and so on, queue calling may not be the best solution, but it would be great if we could queue it directly while sending the e-mails. Laravel's `Email` class also handles this. Instead of using `Mail::send()`, if we use `Mail::queue()` with the same parameters, the e-mail sending will be done with the help of the queue driver, and the response times for the end-user will be faster.

Summary

Throughout this chapter, we've created a simple newsletter subscription form using Laravel's `Form Builder` class using jQuery's Ajax post methods. We've validated and processed the form and saved the data into the database. We've also learned how to queue long processes easily with Laravel's `queue` class using a third-party queue driver. We've also covered the basics of e-mail sending with Laravel 4.

In the next chapter, we will be writing a **Q&A** site, which will have a pagination system, a tag system, a third-party authentication library, a question and answer voting system, options to choose the best answer, and a search system for questions.

8
Building a Q&A Web Application

In this chapter we are going to create a Q&A web application. First, we will learn to remove the public segment from Laravel, to be able to use some shared hosting solutions. Then, we will use a third-party extension for authentication and process access rights. Finally, we will make a question system which will allow commenting and answering questions, a tag system, upvoting and downvoting, and choosing the best answer. We will use pivot tables for question tags. We will also benefit from the jQuery Ajax requests at various places. The following are the topics that will be covered in this chapter:

- Removing the public segment from Laravel 4
- Installing Sentry 2 and an authentication library, and setting access rights
- Creating custom filters
- Creating our registration and login forms
- Creating our questions table and model
- Creating our tags table with a pivot table
- Creating and processing our question form
- Creating our questions list page
- Creating our question page
- Creating our answers table and resources
- Searching questions by tags

Removing the public segment from Laravel 4

In some real-world cases, you may have to stick with badly configured, shared web hosting solutions, which don't have a `www`, `public_html`, or a similar folder. In that case, you would want to remove the public segment from your Laravel 4 installation. To remove this public segment, there are some easy steps to follow:

1. First, make sure you have a running Laravel 4 instance.
2. Then, move everything inside the `public` folder into the parent folder (where `app`, `bootstrap`, `vendor`, and other folders are present), and then delete the blank public folder.
3. Next, open the `index.php` file (which we had just moved from the public folder), and find the following line:

   ```
   require __DIR__.'/../bootstrap/autoload.php';
   ```

   Replace the previous line with the following line:

   ```
   require __DIR__.'/bootstrap/autoload.php';
   ```

4. Now, find this line in the `index.php` file:

   ```
   $app = require_once __DIR__.'/../bootstrap/start.php';
   ```

   Replace the previous line with the following line:

   ```
   $app = require_once __DIR__.'/bootstrap/start.php';
   ```

5. Now, open the `paths.php` file under the `bootstrap` folder, and find this line:

   ```
   'public' => __DIR__.'/../public',
   ```

   Replace the previous line with the following line:

   ```
   'public' => __DIR__.'/..',
   ```

6. If you are using a virtual host, don't forget to change the directory settings and restart your web server.

In the previous steps, we first moved everything from the `public` folder to the `parent` folder since we won't be using the `parent` segment anymore. Then we altered the `index.php` file to identify the proper paths of `autoload.php` and `start.php`, so that the framework can run. If everything went okay, you won't see any issues whatsoever when you refresh your page, and this means you have successfully removed the public segment from your Laravel 4 installation.

Not to forget that this method will make all your code available in the public web root, and this may bring security issues on your project. In that case, you should prevent using this method, or you should find a better web hosting solution.

Installing Sentry 2 and an authentication library and setting access rights

In this section, we will be installing a third-party library for user authentication and access rights called Sentry 2, made available by **Cartalyst**. Cartalyst is a developer centric, open source company with a strong focus on documentation, community support, and framework. In this section, we will be following the Sentry's official Laravel 4 installation steps, with a simple extra step, which is currently available at `http://docs.cartalyst.com/sentry-2/installation/laravel-4`.

1. First, open your `composer.json` file, and add the following line to the `require` attribute:

   ```
   "cartalyst/sentry": "2.0.*"
   ```

2. Then, run the composer update command to fetch the package:

   ```
   php composer.phar update
   ```

3. Now, open your `app.php` file under `app/config`, and add the following line to the `providers` array:

   ```
   'Cartalyst\Sentry\SentryServiceProvider',
   ```

4. Now, add the following line to your `aliases` array in `app.php`:

   ```
   'Sentry' => 'Cartalyst\Sentry\Facades\Laravel\Sentry',
   ```

5. Now, run the following command to install the required tables (or users) to the database:

   ```
   php artisan migrate --package=cartalyst/sentry
   ```

6. Next, we need to publish the configuration file of Sentry 2 to our `app` folder, so that we can manage throttling or other settings if we want to. Run the following command from your terminal:

   ```
   php artisan config:publish cartalyst/sentry
   ```

7. Now, we should alter the default User model to be able to use it with Sentry 2. Open your `User.php` file under `app/models`, and replace all its contents with the following code:

```
<?php
class User extends Cartalyst\Sentry\Users\Eloquent\User {
}
```

8. Lastly, we should create our admin user. Add the following code to your `routes.php` file under the `app` folder and run it once. Comment out or delete the code after that. We are practically assigning the admin ID=1 for our system, with an access right called `admin`.

```
/**
* This method is to create an admin once.
* Just run it once, and then remove or comment it out.
**/
Route::get('create_user',function(){

$user = Sentry::getUserProvider()->create(array(
  'email' => 'admin@admin.com',
  //password will be hashed upon creation by Sentry 2
  'password' => 'password',
  'first_name' => 'John',
  'last_name' => 'Doe',
  'activated' => 1,
  'permissions' => array (
    'admin' => 1
  )
));
return 'admin created with id of '.$user->id;
});
```

   Doing this, you've successfully created a user with `admin@admin.com` as an e-mail address and `password` as the password. The password will be automatically hashed upon creation by Sentry 2, so we won't have to hash and salt the password before creation. We have set the admin's name as `John` and the surname as `Doe`. Also, we've set a permission for the user that we've just generated called `admin` to check the access right before the request processes.

You're now all set. If everything went okay and you check your database, you should see the migrations table generated by Laravel 4 (which you had to manually set before the first migration in Laravel 3) and the tables generated by Sentry 2. In the `users` table, you should see an entry for the user generated by our closure method.

Now that our user authentication system is ready, we need to generate our filters, and then create registration and login forms.

Creating custom filters

Custom filters will help us filter requests and help us make some prechecks beforehand. Benefiting from the Sentry 2's built-in methods, we can define custom filters easily. But first we need to define some routes, which will be used in our project.

Add the following code to your `routes.php` file under the `app` folder:

```
//Auth Resource
Route::get('signup',array('as'=>'signup_form', 'before'=>
  'is_guest', 'uses'=>'AuthController@getSignup'));
Route::post('signup',array('as'=>'signup_form_post', 'before' =>
  'csrf|is_guest', 'uses' => 'AuthController@postSignup'));
Route::post('login',array('as'=>'login_post', 'before' =>
  'csrf| is_guest', 'uses' => 'AuthController@postLogin'));
Route::get('logout',array('as'=>'logout', 'before'=>'
  user', 'uses' => 'AuthController@getLogout'));

//---- Q & A Resources
Route::get('/',array('as'=>'index','uses'=>
  'MainController@getIndex'));
```

In these named resources, the names are defined with the key `as` in the array, and the filters are set with the key `before`. As you can see, there are some `before` parameters, such as `is_guest` and `user`. These filters will run before any request is made by the user, and even call the controller. The key `uses` sets the controller that will be executed when the resource is called. We will write the code for those controllers later. As a result, for example, a user can't even try to post to the login form. If the user tries that, our filter will run and do the filtering before the request is made by the user.

Now that our routes are ready, we can add the filters. To add the filters, open your `filters.php` file under the `app` folder and add the following code:

```
/*
 |--------------------------------------------------------------
 | Q&A Custom Filters
 |--------------------------------------------------------------
*/
```

```
Route::filter('user',function($route,$request){
  if(Sentry::check()) {
    //is logged in
  } else {
    return Redirect::route('index')
      ->with('error','You need to log in first');
  }
});

Route::filter('is_guest',function($route,$request){
  if(!Sentry::check()) {
    //is a guest
  } else {
    return Redirect::route('index')
      ->with('error','You are already logged in');
  }
});

Route::filter('access_check',function($route,$request,$right){
  if(Sentry::check()) {
    if(Sentry::getUser()->hasAccess($right)) {
      //logged in and can access
    } else {
      return Redirect::route('index')
        ->with('error',
          'You don\'t have enough priviliges to access that
          page');
    }
  } else {
    return Redirect::route('index')
      ->with('error','You need to log in first');
  }
});
```

The method `Route::filter()` allows us to create our own filters. The first parameter is the filter's name and the second parameter is a closure function, which itself takes at least two parameters. If you need to provide a parameter to the filter, you can add this as a third parameter.

The `check()` helper function of Sentry 2 returns a Boolean value whether the user is logged in or not. If it returns true, it means the user is logged in, else the user browsing the web page is currently not logged in. In our custom filter `user` and `is_guest`, we check exactly this. The passing clause of your filter can be left blank. But if the user fails to satisfy the filter's conditions, appropriate action can be taken. In our example, we are redirecting the user to our `index` route.

However, our third filter `access_check` is a little bit more complicated. As you can see, we've added a third parameter called `$right`, which we will pass through the calling filter. This filter checks two conditions. First, it checks whether the user is logged in by using the `Sentry::check()` method. Then, it checks whether the user has access to the `$right` section (which we will see when we define filters) by using the `hasAccess()` method. But this method requires a current logged in user first. For this, we will validate the current user's information by using the `getUser()` method of Sentry 2.

To pass parameter(s) while calling a filter, you can use `filter_name:parameter1, parameter2`. In our example, we will check whether the user is an admin, using the filter `access_check:admin`.

To use multiple filters in the `before` parameter, add a `|` character between the parameters. In our example, our login post and sign up resources' filters are defined as `csrf|guest` (csrf is predefined in our `filters.php` file by Laravel itself).

Creating our registration and login forms

Before creating our registration and login forms, we need a template to set the sections. I'll be using a custom HTML/CSS template that I've generated for this chapter, which is inspired by the **Snow** theme of the open source Q&A script, **Question2Answer**.

We perform the following steps to create our registration and login forms:

1. First, copy everything in the `assets` folder of the provided example code, to your project folder's root (where the `app`, `bootstrap`, and other folders are located), because we had removed the public folder segment in the first section of this chapter.
2. Next, add the following code to your `template_masterpage.blade.php` file under `app/views`:

```
<!DOCTYPE html>
<!--[if lt IE 7]> <html class="no-js lt-ie9 lt-ie8 lt-ie7">
<![endif]-->
<!--[if IE 7]> <html class="no-js lt-ie9 lt-ie8">
<![endif]-->
<!--[if IE 8]> <html class="no-js lt-ie9">
<![endif]-->
<!--[if gt IE 8]><!--> <html class="no-js">
<!--<![endif]-->

<head>
  <meta charset="utf-8" />
```

```
    <title>{{isset($title)?$title.' | ':''}} LARAVEL Q & A
    </title>
    {{ HTML::style('assets/css/style.css') }}
  </head>
  <body>

    {{-- We include the top menu view here --}}
    @include('template.topmenu')

    <div class="centerfix" id="header">
    <div class="centercontent">
      <a href="{{URL::route('index')}}">
        {{HTML::image('assets/img/header/logo.png')}}
      </a>
    </div>
    </div>
    <div class="centerfix" id="main" role="main">
    <div class="centercontent clearfix">
      <div id="contentblock">

      {{-- Showing the Error and Success Messages--}}
      @if(Session::has('error'))
      <div class="warningx wredy">
        {{Session::get('error')}}
      </div>
      @endif

      @if(Session::has('success'))
      <div class="warningx wgreeny">
        {{Session::get('success')}}
      </div>
      @endif

      {{-- Content section of the template --}}
      @yield('content')
      </div>
    </div>
    </div>
    {{-- JavaScript Files --}}
    {{ HTML::script('assets/js/libs.js') }}
    {{ HTML::script('assets/js/plugins.js') }}
    {{ HTML::script('assets/js/script.js') }}
```

```
  {{-- Each page's custom assets (if available) will be
    yielded here --}}
  @yield('footer_assets')

</body>
</html>
```

Now, let's dig the code:

- If we load a view with a `title` attribute, the `<title>` tag will include the title; else it will just display our website's name.
- The `style()` method of the `HTML` class will help us add CSS files to our template easily. Also, the `script()` method of the `HTML` class allows us to add JavaScript to our output HTML file.
- We have included another file to our `template_masterpage.blade.php` file using the `@include()` method of the Blade template engine. We will describe its sections in the next step.
- The `route()` method of the `URL` class will return a link to a named route. This is pretty handy actually, because if we change the URL structure we won't need to dig into all the template files and edit all our links.
- The `image()` method of the `HTML` class allows us to add the `<img>` tag to our template.
- In filters, we redirected to the route pages using the `with()` method with the parameter `error`. If we loaded pages (`View::make()`) using `with()`, the parameters would be variables. But because we have redirected the user to a page, these parameters passed with `with()` will be a session flashdata, which will only be available once. To check whether these sessions are set, we use the `has()` method of the `Session` class. `Session::has('sessionName')` will return a Boolean value to identify whether a session is set or not. If it's set, we can use the `get()` method of the `Session` class to use it in our views, controllers, and other places as well.
- The `@yield()` method of the Blade template engine fetches the data present in `@section()`, and parses it to the master template page.

3. In the previous section, we included another view by calling the `@include()` method as `@include('template.topmenu')`. Now save the following code as `topmenu.blade.php` under `app/views/template`:

```
{{-- Top error (about login etc.) --}}
@if(Session::has('topError'))
  <div class="centerfix" id="infobar">
```

```
    <div class="centercontent">{{ Session::get
      ('topError') }}
    </div>
  </div>
@endif

{{-- Check if a user is logged in, login and logout has
  different templates --}}
@if(!Sentry::check())
<div class="centerfix" id="login">
  <div class="centercontent">
    {{Form::open(array('route'=>'login_post'))}}
    {{Form::email('email', Input::old('email'),
      array('placeholder'=>'E-mail Address'))}}
    {{Form::password('password', array('placeholder' =>
      'Password'))}}
    {{Form::submit('Log in!')}}
    {{Form::close()}}

    {{HTML::link('signup_form','Register',array(),
      array('class'=>'wybutton'))}}
  </div>
</div>
@else
  <div class="centerfix" id="login">
    <div class="centercontent">
      <div id="userblock">Hello again,
        {{HTML::link('#',Sentry::getUser()->first_name.'
        '.Sentry::getUser()->last_name)}}</div>
      {{HTML::linkRoute('logout','Logout',array(),
        array('class'=>'wybutton'))}}
    </div>
  </div>
@endif
```

Now, let's dig the code:

- In our template we have two error messages of which the first one is totally reserved for the login area that will be shown at the top. I've named it as `error_top`. With the methods `has()` and `get()` that we've just learned, we check whether an error is present, and display it.
- The top menu will depend on whether a user is logged in or not. So we create an `if` clause using the user checking method `check()` of Sentry 2 and check if the user is logged in. If the user is not logged in (guest), we show the login form that we've made using the `Form` class, else we show the user `infobar` with a profile and a **logout** button.

4. Now, we need a registration form page. We've already defined its methods in our `routes.php` file under the `app` folder earlier:

```
//Auth Resource
Route::get('signup',array('as'=>'signup_form',
  'before' => 'is_guest',
  'uses' => 'AuthController@getSignup'));
Route::post('signup',array('as' => 'signup_form_post',
  'before' => 'csrf|is_guest',
  'uses' => 'AuthController@postSignup'));
```

5. According to the route resource we've created, we need a controller named `AuthController`, having two methods called `getSignup()` and `postSignup()`. Now let's first create the controller. Open your terminal and type the following command:

```
php artisan controller:make AuthController
```

6. The previous command will create a new file, `AuthController.php` under `app/controllers` with some default methods. Delete the code present inside the `AuthController` class and add the following code inside that class, to make the sign up form:

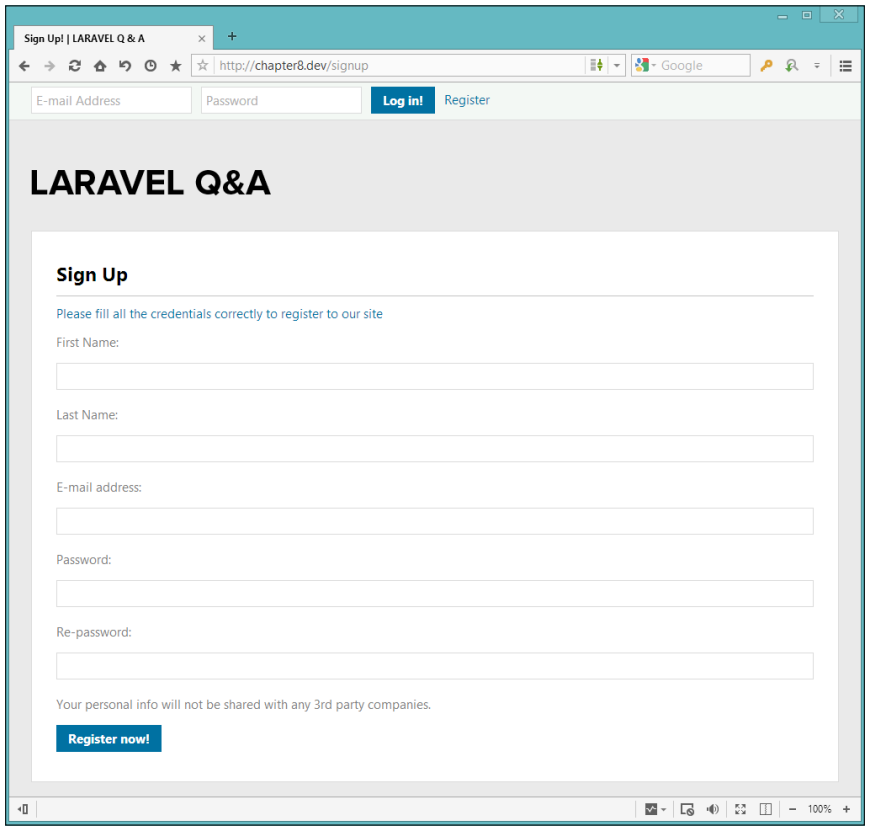
```
/**
  * Signup GET method
**/
public function getSignup() {
  return View::make('qa.signup')
    ->with('title','Sign Up!');
}
```

7. We now need a view file to make the form. Save the following code as `signup.blade.php` under `app/views/qa`:

```
@extends('template_masterpage')

@section('content')
  <h1 id="replyh">Sign Up</h1>
  <p class="bluey">Please fill all the credentials
    correctly to register to our site</p>

  {{Form::open(array('route'=>'signup_form_post'))}}
    <p class="minihead">First Name:</p>
    {{Form::text('first_name',Input::get('first_name'),
      array('class'=>'fullinput'))}}
    <p class="minihead">Last Name:</p>
```

```
        {{Form::text('last_name',Input::get('last_name'),
          array('class'=>'fullinput'))}}
        <p class="minihead">E-mail address:</p>
        {{Form::email('email',Input::get('email'),
          array('class'=>'fullinput'))}}
        <p class="minihead">Password:</p>
        {{Form::password('password','',
          array('class'=>'fullinput'))}}
        <p class="minihead">Re-password:</p>
        {{Form::password('re_password','',
          array('class'=>'fullinput'))}}
        <p class="minihead">Your personal info will not be
          shared with any 3rd party companies.</p>
        {{Form::submit('Register now!')}}
      {{Form::close()}}
@stop
```

If you have done everything correctly, when you navigate to `chapter8.dev/signup`, you should see the following form:



Validating and processing the form

Now, we need to validate and process the form. We first need to define our validation rules. Add the following code to the `User` class in your `user.php` file under `app/models`:

```
public static $signup_rules = array(
  'first_name' => 'required|min:2',
  'last_name' => 'required|min:2',
  'email' => 'required|email|unique:users,email',
  'password' => 'required|min:6',
  're_password' => 'required|same:password'
);
```

The rules mentioned in the previous code will make all the fields `required`. We set the `first_name` and `last_name` columns as `required`, and we set a minimum length of two characters. We set the `email` field to be in a valid e-mail format, and the code will check the `users` table (which is created upon installing Sentry 2) for unique e-mail addresses. We set the `password` field to be `required`, and its length should be a minimum of six characters. We also set the `re_password` field to match the `password` field, so that we can make sure that the password is typed correctly.

Sentry 2 can also throw a unique e-mail checking exception, upon an attempt to log in a user.

Before processing the form, we need a dummy index page to return the user after signing up successfully. We will create a temporary index page by performing the following steps:

1. First, run the following command to create a new controller:

   ```
   php artisan controller:make MainController
   ```

2. Then, remove all methods that auto inserted, and add the following method inside the class:

   ```
   public function getIndex() {
     return View::make('qa.index');
   }
   ```

3. Now, save this view file as `index.blade.php` under `app/views/qa`:

   ```
   @extends('template_masterpage')

   @section('content')
   Heya!
   @stop
   ```

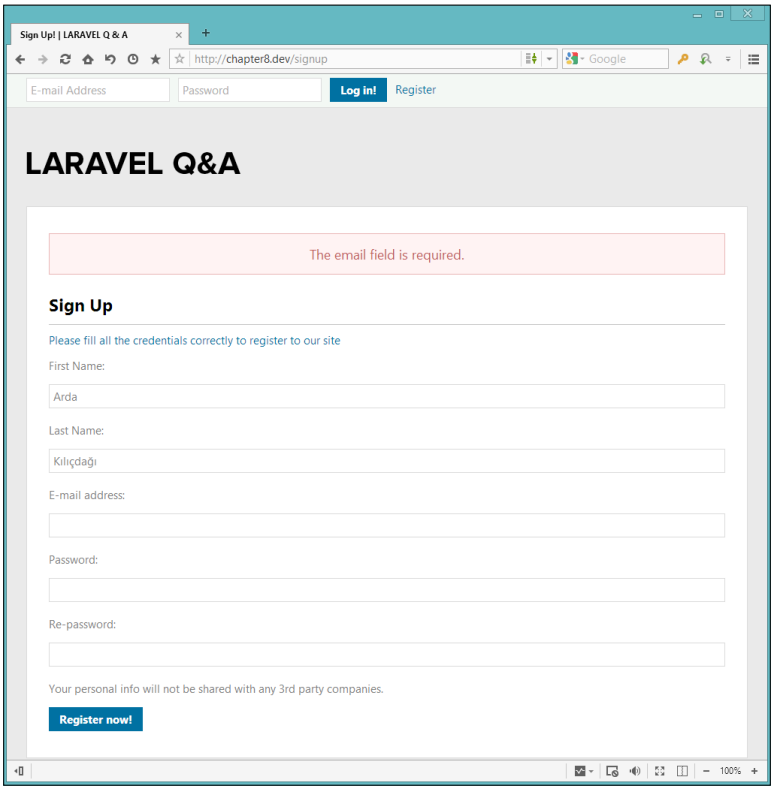
4. Now, we need a controller method (which we defined in `routes.php`) to process the `signup` form's `post` request. To do this, add the following code to your `AuthController.php` file under `app/controllers`:

```
/**
  * Signup Post Method
**/
public function postSignup() {

  //Let's validate the form first
  $validation = Validator::make(Input::all(),User::$signup_rules);

  //let's check if the validation passed
  if($validation->passes()) {

    //Now let's create the user with Sentry 2's create method
    $user = Sentry::getUserProvider()->create(array(
      'email' => Input::get('email'),
      'password' => Input::get('password'),
      'first_name' => Input::get('first_name'),
      'last_name' => Input::get('last_name'),
      'activated' => 1
    ));

    //Since we don't use an email validation in this example,
      let's log the user in directly
    $login = Sentry::authenticate(array(
      'email'=>Input::get('email'),
      'password'=>Input::get('password')));

    return Redirect::route('index')
      ->with('success','You\'ve signed up and logged in
        successfully!');
    //if the validation failed, let's return the user
    //to the signup form with the first error message
  } else {
    return Redirect::route('signup_form')
    ->withInput(Input::except('password','re_password'))
      ->with('error',$validation->errors()->first());
  }
}
```

Now, let's dig the code:

1. First, we check the form items using Laravel's built-in form validation class using the rules we've defined in the model.
2. We check whether the form validation passes, using the `passes()` method. We could also check the exact opposite situation using the `fails()` method.

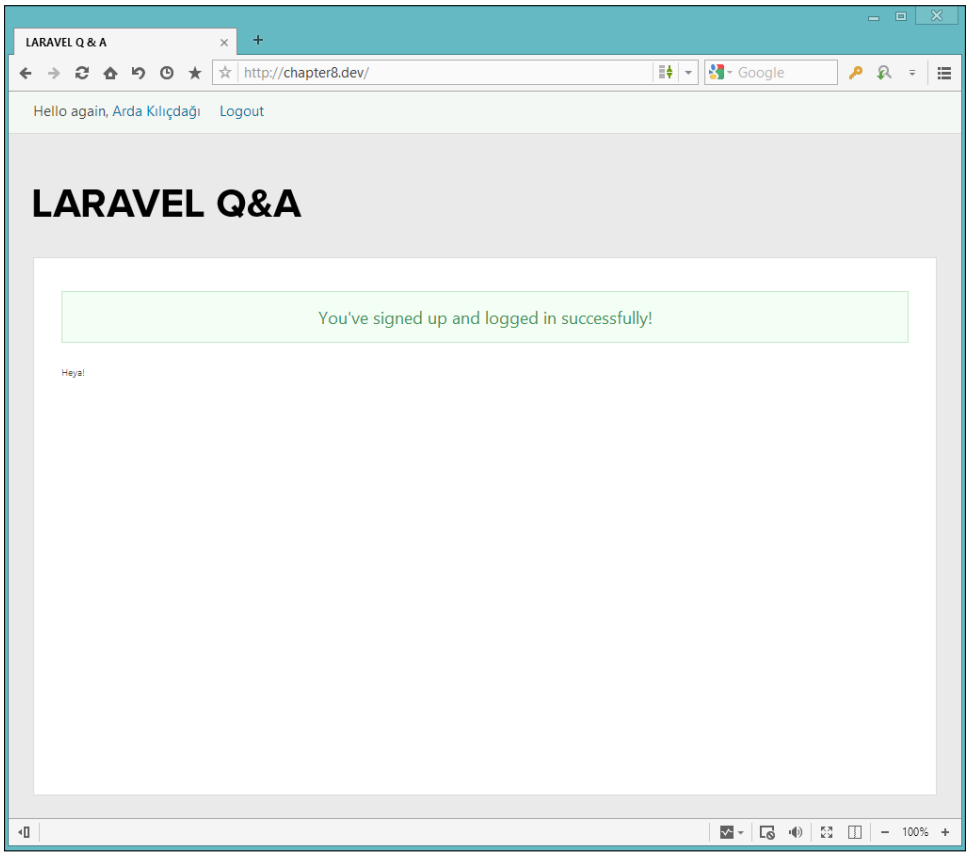
If the validation fails, we return the user to the **Sign Up** form with given credentials using `withInput()`. But by using `Input::except()`, we filter some columns such as `password` and `re_password`, so that the value in those fields are not returned. Also, by passing a parameter using `with`, the form validation's error message is returned. `$validation->errors()->first()` returns the first error message string after the form validation step.



If the validation passes, we create a new user using the provided credentials. We have set the column `activated` to `1`, so that the sign up process would not require an e-mail validation in our example.

Sentry 2 also uses a try/catch clause to catch errors. Don't forget to check the documentation of Sentry 2, to learn how to catch unusual errors.

1. Since we are not using an e-mail validation system, we could simply authenticate and sign in the user using the `authenticate()` method of Sentry 2, right after signing up. The first parameter takes an array of `email` and `password` (with `key => value` matching) and the optional second parameter takes a Boolean value as an input, to check whether the user is to be remembered or not (the `remember me` button).
2. After the authentication, we simply redirect the user to our `index` route with a success message, as shown in the following screenshot:



Processing the login and logout requests

Now that our registration system is ready, we need to process login and logout requests. Since our login form is already prepared, we can directly go ahead and process it. To process login and logout requests, we perform the following steps:

1. First, we need the login form validation rules. Add the following code to your `User.php` file under `app/models`:

```
public static $login_rules = array(
       'email'             => 'required|email|exists:users,email',
       'password'    => 'required|min:6'
);
```

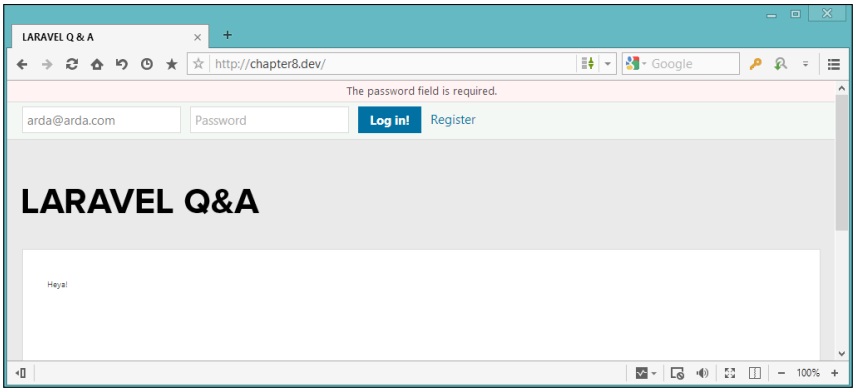
2. Now, we need a controller method to process the login request. Add the following code to your `AuthController.php` file under `app/controllers`:

```
/**
 * Login Post Method Resource
**/
public function postLogin() {
  //let's first validate the form:
  $validation =
    Validator::make(Input::all(),User::$login_rules);

  //if the validation fails, return to the index page with
    first error message
  if($validation->fails()) {
    return Redirect::route('index')
      ->withInput(Input::except('password'))
      ->with('topError',$validation->errors()->first());
  } else {

    //if everything looks okay, we try to authenticate the
      user
    try {

      // Set login credentials
      $credentials = array(
        'email' => Input::get('email'),
        'password' => Input::get('password'),
      );

      // Try to authenticate the user, remember me is set
        to false
      $user = Sentry::authenticate($credentials, false);
      //if everything went okay, we redirect to index route
        with success message
```

```
            return Redirect::route('index')
              ->with('success',
                'You\'ve successfully logged in!');
          } catch (Cartalyst\Sentry\Users\
            LoginRequiredException $e) {
            return Redirect::route('index')
              -> withInput(Input::except('password'))
              ->with('topError','Login field is required.');
          } catch (Cartalyst\Sentry\Users\
            PasswordRequiredException $e) {
            return Redirect::route('index')
              -> withInput(Input::except('password'))
              ->with('topError','Password field is required.');
          } catch (Cartalyst\Sentry\Users\
            WrongPasswordException $e) {
            return Redirect::route('index')
              -> withInput(Input::except('password'))
              ->with('topError','Wrong password, try again.');
          } catch (Cartalyst\Sentry\Users\
            UserNotFoundException $e) {
            return Redirect::route('index')
              -> withInput(Input::except('password'))
              ->with('topError','User was not found.');
          } catch (Cartalyst\Sentry\Users\
            UserNotActivatedException $e) {
            return Redirect::route('index')
              -> withInput(Input::except('password'))
              ->with('topError','User is not activated.');
          }

          // The following is only required if throttle is
            enabled
          catch (Cartalyst\Sentry\Throttling\
            UserSuspendedException $e) {
          return Redirect::route('index')
            -> withInput(Input::except('password'))
            ->with('topError','User is suspended.');
          } catch (Cartalyst\Sentry\Throttling\
            UserBannedException $e) {
            return Redirect::route('index')
              -> withInput(Input::except('password'))
              ->with('topError','User is banned.');
          }
        }
      }
```

Now, let's dig the code:

1. First, we check the form items via Laravel's built-in form validation class, using the rules we've defined in the model.
2. Then we check whether the form validation has failed using the `fails()` method of the form validation class. If the form validation fails, we return the user to the `index` route with the first form validation error.



3. The `else` clause in the previous code holds the events that will be done if the form validation passes. In this, we authenticate a user using the try/catch clauses of Sentry 2, catch all the exceptions, and return an error message according to the type of exception.

We don't need all the exceptions in our example application, but as an example we tried to show all the exceptions, just in case you needed to do something different while following up.

All these try/catch exceptions are documented on the website of Sentry 2.

4. If no exceptions were thrown by Sentry 2, we return to the index page with a success message.

3. Now, the only thing remaining regarding authentication is the logout button. To create one, add the following code to the `AuthController.php` file under `app/controllers`:

```
/**
  * Logout method
**/
```

```
public function getLogout() {
  //we simply log out the user
  Sentry::logout();

  //then, we return to the index route with a success
    message
  return Redirect::route('index')
    ->with('success','You\'ve successfully signed out');
}
```

Now let's dig the code:

1. First, we call the `logout()` method of Sentry 2, which logs the user out.
2. Then, we simply return the user (who is currently a guest) to the `index` route with a success message, telling that they have successfully logged out.

Now that our authentication system is ready, we are ready to create our questions table.

Creating our questions table and model

Now that we have a fully working authentication system, we are ready to create our `questions` table. To create our `questions` table, we will be using a database migration.

To create a migration, run the following command in your terminal:

```
php artisan migrate:make create_questions_table --table=
  questions --create
```

The previous command will create a new migration under `app/database/migrations`.

For questions, we will be needing a question title, question details, the question's poser, the question's date, how many times a question has been viewed, total sum of votes, and the question's tags.

Now, open the migration that you've just created and replace its content with the following code:

```
Schema::create('questions', function(Blueprint $table)
{
  //Question's ID
  $table->increments('id');
```

```
    //title of the question
    $table->string('title',400)->default('');
    //asker's id
    $table->integer('userID')->unsigned()->default(0);
    //question's details
    $table->text('question')->default('');
    //how many times it's been viewed:
    $table->integer('viewed')->unsigned()->default(0);
    //total number of votes:
    $table->integer('votes')->default(0);
    //Foreign key to match userID (asker's id) to users
    $table->foreign('userID')->references('id')->
      on('users')->onDelete('cascade');
    //we will get asking time from the created_at column
    $table->timestamps();
});
```

For tags, we will be using a pivot table, that's why they are not present in our current schema. For votes, in this example, we are simply holding an integer (that can be positive or negative). In a real-world application, you would want to use a second pivot table to keep users' votes, to prevent double voting, and to get a more accurate result.

1. Now that your schema is ready, run the migration using the following command:

   ```
   php artisan migrate
   ```

2. After successfully migrating the schema, we now need a model to benefit from Eloquent. Save the following code as `Question.php` under `app/models`:

   ```
   <?php

   class Question extends Eloquent {

     protected $fillable = array('title', 'userID', 'question',
       'viewed', 'answered', 'votes');

   }
   ```

3. Now, we need the database relations to match tables. First, add the following code to your `User.php` file under `app/models`:

   ```
   public function questions() {
     return $this->hasMany('Question','userID');
   }
   ```

4. Next, add the following code to your `Question.php` file under `app/models`:

```
public function users() {
  return $this->belongsTo('User','userID');
}
```

Since a user may have more than one question, we have used the `hasMany()` method for the relation in our `User` model. Also, since all the questions are owned by the users, we have used the `belongsTo()` method to match questions to users. In these methods, the first parameter is the model name, which in our case is `Question` and `User`. The second parameter is the column name in that model to match the tables, which in our case is `userID`.

Creating our tags table with a pivot table

First, we should understand why we need pivot tables for tags. In a real world situation, a question may have more than one tag; also, a tag may have more than one question. In such situations (many to many relationships), where both the tables may have more than one of each other to match them properly, we should create and use a third pivot table.

1. First, we should create a new tags table using schema. Open your terminal and run the following command to create our pivot table schema:

```
php artisan migrate:make create_tags_table --table= tags --create
```

2. Now we need to fill the table's contents. In our example, we just need the tag name and tag's friendly URL name. Replace the schema's `up` function contents with the following code:

```
Schema::create('tags', function(Blueprint $table)
{
  //id is needed to match pivot
  $table->increments('id');

  //Tag's name
  $table->string('tag')->default('');
  //Tag's URL-friendly name
  $table->string('tagFriendly')->unique();

  //I like to keep timestamps
  $table->timestamps();
});
```

We have the `id` column to match questions with tags in the pivot table. We have a string field `tag`, which will be the title of the tag, and the column `tagFriendly` is what will be shown as a URL. I have also kept timestamps, so that, in future, it can give us information about when the tag was created.

3. Lastly, run the following command in your terminal to run the migration and install the table:

```
php artisan migrate
```

4. Now, we need a model for the `tags` table. Save the following file as `Tag.php` under `app/models`:

```
<?php

class Tag extends Eloquent {

  protected $fillable = array('tag', 'tagFriendly');

}
```

5. Now, we need to create our pivot table. As a good practice, its name should be `modelname1_modelname2`, and its content sorted alphabetically. In our example, we have the `questions` and `tags` table, so we will set the pivot table's name as `question_tags` (this is not forced, you can give any name to your pivot table). As you may guess, its schema will have two columns to match the two tables and two foreign keys for these columns. You can even add additional columns to the pivot table.

   To create the migration file, run the following command in your terminal:

```
php artisan migrate:make create_question_tags_table
--table=question_tags --create
```

6. Now, open the schema that we've generated in the `migrations` folder under `app/database` and alter its `up()` method contents with the following code:

```
Schema::create('question_tags', function(Blueprint $table)
{
  $table->increments('id');

  $table->integer('question_id')->unsigned()->default(0);
  $table->integer('tag_id')->unsigned()->default(0);

  $table->foreign('question_id')->references('id')->
    on('questions')->onDelete('cascade');
  $table->foreign('tag_id')->references('id')->
    on('tags')->onDelete('cascade');
```

```
  $table->timestamps();
});
```

We need two columns, and its name structure should be `modelname_id`. In our migration, they are `question_id` and `tag_id`. Also, we've set the foreign keys to match them in our database.

7. Now, run the migration and install the table:

```
php artisan migrate
```

8. Now, we need to add methods to describe to Eloquent that we are using a pivot table. To teach the pivot information to the question model, add the following code to the `Question.php` file under `app/models`:

```
public function tags() {
  return $this->belongsToMany('Tag','question_tags')->
    withTimestamps();
}
```

To describe the pivot information to the tag model, add the following code to the `Tag.php` file under `app/models`:

```
public function questions() {
  return $this->
    belongsToMany('Question','question_tags')->
    withTimestamps();
}
```

The first parameter in the `belongsToMany()` method is the model name, and the second parameter is the pivot table's name. Using `withTimestamps()` (which brings us the pivot data's creation and updation dates) is optional. Also, if we had some extra data to be added to the pivot table, we could call it using the method `withPivot()`. Consider the following example code:

```
$this->belongsToMany('Question ', 'question_tags')->
  withPivot('column1', 'column2')->withTimestamps();
```

Now that our pivot table structure is ready, in the later chapters, we can easily fetch both the question's tags and all questions tagged with `$tagname` easily.

Creating and processing our question form

Now that our structure is ready, we can proceed to create and process our question form.

Creating our questions form

We perform the following steps to create our question form:

1. First, we need a new route resource for the question form. Open your `routes.php` file in the `app` folder and add the following code:

```
Route::get('ask',array('as'=>'ask', 'before'=>'user',
  'uses' => 'QuestionsController@getNew'));

Route::post('ask',array('as'=>'ask_post',
  'before'=>'user|csrf', 'uses' =>
  'QuestionsController@postNew'));
```

2. Now that our resource is defined, we need to add the resource to our top menu for navigation. Open your `topmenu.blade.php` file under `app/views/template`, and find the following line:

```
{{HTML::linkRoute('logout','Logout',array(),
  array('class'=>'wybutton'))}}
```

   Add the previous line above the following line:

```
{{HTML::linkRoute('ask','Ask a Question!', array(),
  array('class'=>'wybutton'))}}
```

3. Now, we need the controller file to handle the resources. Run the following command in your terminal:

```
php artisan controller:make QuestionsController
```

4. Next, open the newly created `QuestionsController.php` file under `app/controllers` and delete all the methods inside the class. Then add the following code:

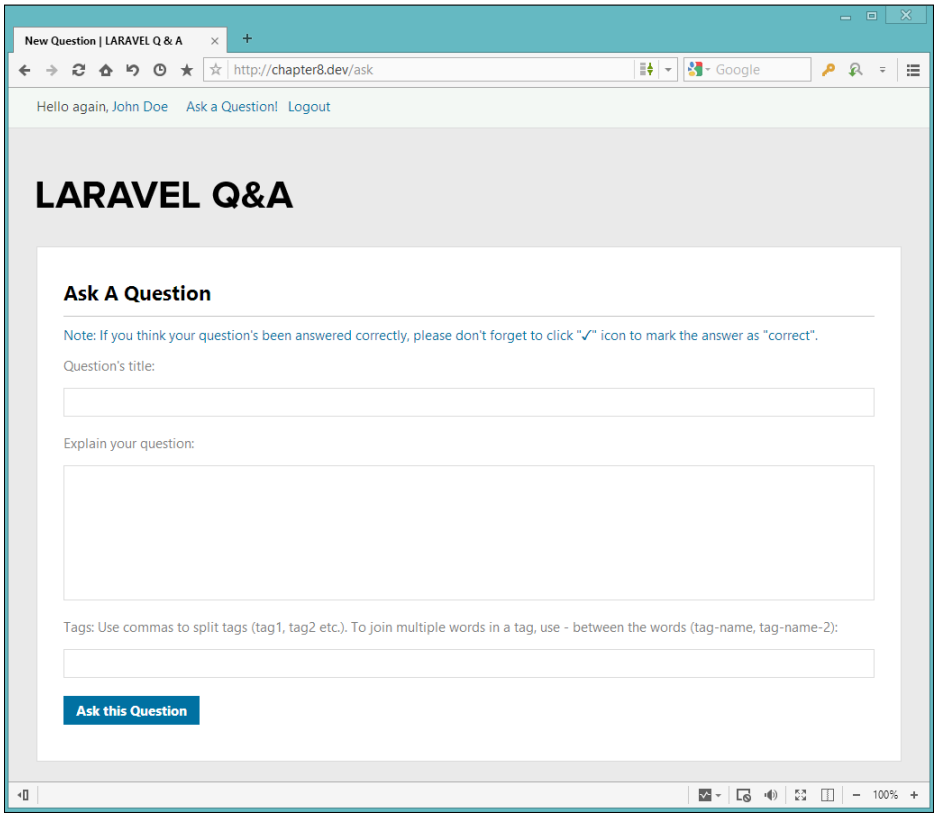
```
/**
  * A new question asking form
**/
public function getNew() {
  return View::make('qa.ask')
    ->with('title','New Question');
}
```

5. Now, we need to create the view we've just assigned. Save the following code as `ask.blade.php` under `app/views/qa`:

```
@extends('template_masterpage')

@section('content')

  <h1 id="replyh">Ask A Question</h1>
  <p class="bluey">Note: If you think your question's
    been answered correctly, please don't forget
    to click " " icon to mark the answer as "correct".</p>
  {{Form::open(array('route'=>'ask_post'))}}

  <p class="minihead">Question's title:</p>
  {{Form::text('title',Input::old('title'),array('class'=>
    'fullinput'))}}

  <p class="minihead">Explain your question:</p>
  {{Form::textarea('question',Input::old('question'),
    array('class'=>'fullinput'))}}

  <p class="minihead">Tags: Use commas to split tags
    (tag1, tag2 etc.). To join multiple words in a tag,
    use - between the words (tag-name, tag-name-2):</p>
  {{Form::text('tags',Input::old('tags'),
    array('class'=>'fullinput'))}}
  {{Form::submit('Ask this Question')}}
  {{Form::close()}}

@stop
@section('footer_assets')

  {{-- A simple jQuery code to lowercase all tags
    before submission --}}
  <script type="text/javascript">
    $('input[name="tags"]').keyup(function(){
      $(this).val($(this).val().toLowerCase());
    });
  </script>

@stop
```

In addition to the previous views we had created, in this view we added a JavaScript code to the footer by filling the `footer_assets` section, which we defined earlier in our master page.

6. If you have done everything correctly, when you navigate to `site.com/ask`, you will see a form styled like the following screenshot:



Now that our question form is ready, we are ready to process the form.

Processing our questions form

To process the form, we need some validation rules and the controller method.

1. First, add the following form validation rules to your `Question.php` file under `app/models`:

```
public static $add_rules = array(
  'title' => 'required|min:2',
  'question' => 'required|min:10'
);
```

2. After saving the question successfully, we would like to provide the question's permalink to the user, so the user can access the question easily. But to do this, we first need to define a route to create this link. Add the following line into your `routes.php` file in the `app` folder:

```
Route::get('question/{id}/{title}',array(
  'as'=> 'question_details', '
  uses' => 'QuestionsController@getDetails' ))->
  where(array('id'=>'[0-9]+' ,
  'title' => '[0-9a-zA-Z\-\_]+'));
```

   We set two parameters into this route, `id` and `title`. The `id` parameter has to be a positive integer, whereas `title` should contain only alphanumeric characters, score, and underscore.

3. Now, we are ready to process the question form. Add the following code to your `QuestionsController.php` file under `app/controllers`:

```
/**
 * Post method to process the form
**/
public function postNew() {

  //first, let's validate the form
  $validation = Validator::make(Input::all(),
    Question::$add_rules);

  if($validation->passes()) {
    //First, let's create the question
    $create = Question::create(array(
      'userID' => Sentry::getUser()->id,
      'title' => Input::get('title'),
      'question' => Input::get('question')
    ));

    //We get the insert id of the question
    $insert_id = $create->id;

    //Now, we need to re-find the question to "attach" the
      tag to the question
    $question = Question::find($insert_id);

    //Now, we should check if tags column is filled, and
      split the string and add a new tag and a relation
    if(Str::length(Input::get('tags'))) {
      //let's explode all tags from the comma
      $tags_array = explode(',', Input::get('tags'));
```

```
//if there are any tags, we will check if they are
  new, if so, we will add them to database
//After checking the tags, we will have to "attach"
  tag(s) to the new question
if(count($tags_array)) {
  foreach ($tags_array as $tag) {
    //first, let's trim and get rid of the extra
      space bars between commas
    //(tag1, tag2, vs tag1,tag2)
    $tag = trim($tag);

    //We should double check its length, because the
      user may have just typed "tag1,,tag2"
      (two or more commas) accidentally
    //We check the slugged version of the tag,
      because tag string may only be meaningless
      character(s), like "tag1,+++//,tag2"
    if(Str::length(Str::slug($tag))) {
      //the URL-Friendly version of the tag
      $tag_friendly = Str::slug($tag);

      //Now let's check if there is a tag with the
        url friendly version of the provided tag
          already in our database:
      $tag_check =
        Tag::where('tagFriendly',$tag_friendly);

      //if the tag is a new tag, then we will create
        a new one
      if($tag_check->count() == 0) {
        $tag_info = Tag::create(array(
          'tag' => $tag,
          'tagFriendly' => $tag_friendly
        ));

        //If the tag is not new, this means There was
          a tag previously added on the same name to
          another question previously
        //We still need to get that tag's info from
          our database
      } else {
        $tag_info = $tag_check->first();
      }
    }
```

```
            //Now the attaching the current tag to the
              question
            $question->tags()->attach($tag_info->id);
          }
        }
      }

      //lastly, we should return the user to the asking page
        with a permalink of the question
      return Redirect::route('ask')
        ->with('success','Your question has been created
        successfully! '.HTML::linkRoute('question_details',
        'Click here to see your question',array(
        'id'=>$insert_id,'title'=>Str::slug($question->
        title))));

    } else {
      return Redirect::route('ask')
        ->withInput()
        ->with('error',$validation->errors()->first());
    }
}
```

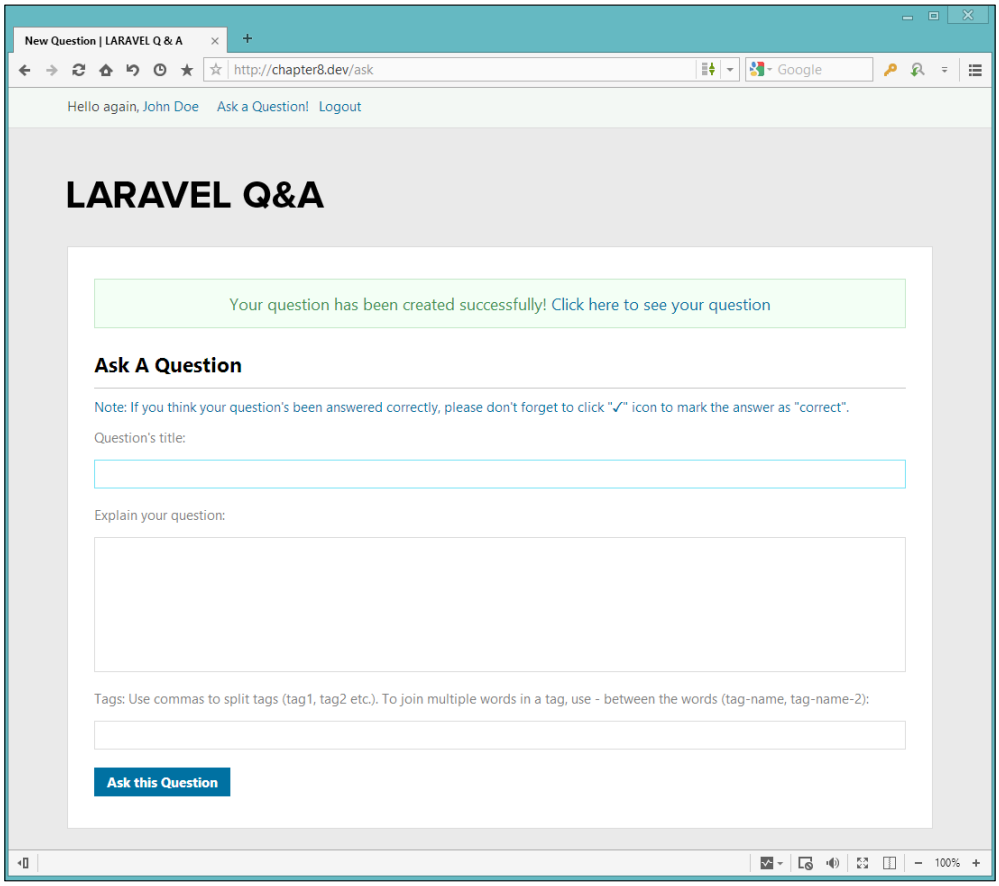
Now, let's dig the code:

1. First, we run the form validation class to check whether the values are valid. If the validation has failed, we return the user to the question page with the old inputs he had provided, and with the first validation error message.
2. If the validation passes, we continue processing the form. We first create and add the question, add a new row to the database, and then we fetch the row that we've just created. To get the current user's ID, we use the `id` object of the `getUser()` method by Sentry 2, which returns the info of the current logged in user.
3. After creating the question, we check the length of the `tags` field. If the field is not empty, we split the string at the commas and make a raw `tags` array.
4. After that, we loop through each of the tags that we had split, and make their friendly URL version using Laravel 4's `slug()` method of the `String` class. If the slugged version has a length, it's a valid tag.

5. After finding all the valid tags, we check the database to find whether there is a tag already created. If so, we get its ID. If the tag is new to the system, then we create a new tag. So, in this way, we avoid unnecessary multiple tags in our system.
6. After that, we use the method `attach()` to create a new tag in the pivot table. To attach a new relation, we first need to find the ID, which we want to attach, and then go to the model of the attachment and use the `attach()` method.
7. In our example, we need to attach the question to the tag(s). So we find the question, which needs to be attached, use a many-to-many relation to show that tags will be attached to the question, and attach the tag's `id` to the question.
8. If everything goes without any problem, you should be redirected back to the question page with a success message and a permalink to your question.
9. Also, if you check your `question_tags` table, you will see the relation data filled.

Always validate and filter the contents coming from forms, and make sure you are not accepting any unwanted content.

After successfully adding the question, you should see a page like the following screenshot:



Creating our questions list page

Now that we can create questions, it's time to fill our dummy index page with actual question data. To do this, open your `MainController.php` file under `app/controllers`, and alter the function `getIndex()` with the following code:

```
public function getIndex() {
  return View::make('qa.index')
    ->with('title','Hot Questions!')
    ->with('questions',Question::with('users','tags')->
      orderBy('id','desc')->paginate(2));
}
```

In this method, we loaded the same page, but we added two variables named `title` and `questions`. The `title` variable is the dynamic title of our application, and the `questions` variable holds the last two questions, with pagination. Instead of `get()`, if you use `paginate($number)`, you can get a ready-to-use pagination system. Also, using the method `with()`, we eagerly loaded the `users` and `tags` relations directly with the questions collection, for better performance.

In the view, we will have a crude upvote/downvote option for the questions, and a route link for the questions tagged with `$tag`. For this, we will need some new routes. Add the following code to your `routes.php` file under the `app` folder:

```
//Upvoting and Downvoting
Route::get('question/vote/{direction}/{id}',array('as'=>
  'vote', 'before'=>'user', 'uses'=>
  'QuestionsController@getvote'))->where
  (array('direction'=>'(up|down)', 'id'=>'[0-9]+'));

//Question tags page
Route::get('question/tagged/{tag}',array('as'=>
  'tagged','uses'=>'QuestionsController@getTaggedWith'))->
  where('tag','[0-9a-zA-Z\-\_]+');
```

Now open your `index.blade.php` file under `app/views/qa`, and alter the whole file with the following code:

```
@extends('template_masterpage')

@section('content')
  <h1>{{$title}}</h1>

  @if(count($questions))

    @foreach($questions as $question)

      <?php
        //Question's asker and tags info
        $asker = $question->users;
        $tags = $question->tags;
      ?>

      <div class="qwrap questions">
        {{-- Guests cannot see the vote arrows --}}
        @if(Sentry::check())
          <div class="arrowbox">
```

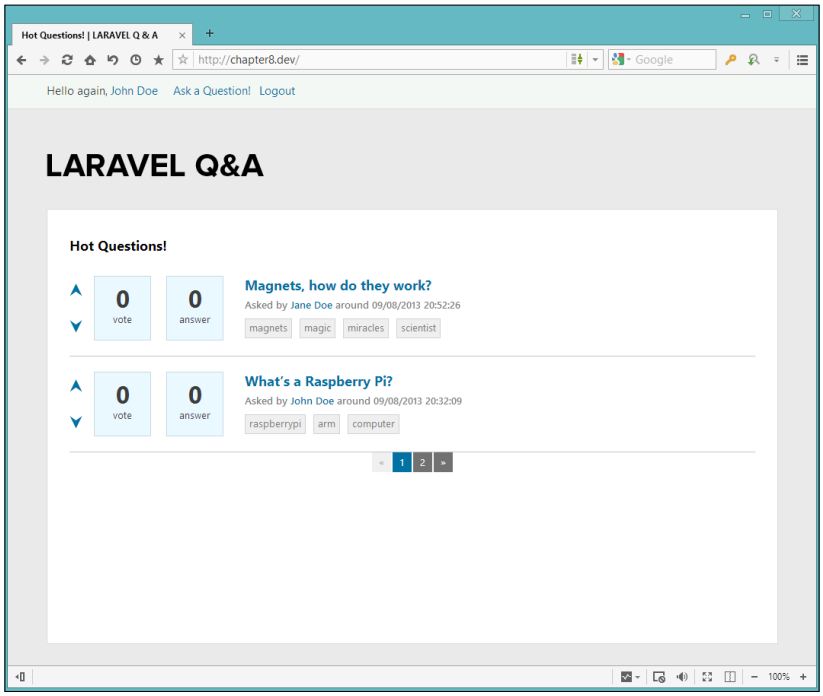
```
    {{HTML::linkRoute('vote','',array('up',
      $question->id),array('class'=>'like',
      'title'=>'Upvote'))}}
    {{HTML::linkRoute('vote','',array('down',
      $question->id),array('class'=>'dislike',
      'title'=>'Downvote'))}}
  </div>
@endif

{{-- class will differ on the situation --}}
@if($question->votes > 0)
  <div class="cntbox cntgreen">
@elseif($question->votes == 0)
  <div class="cntbox">
@else
  <div class="cntbox cntred">
@endif
<div class="cntcount">{{$question->votes}}</div>
<div class="cnttext">vote</div>
</div>

{{--Answer section will be filled later in this
  chapter--}}
<div class="cntbox">
  <div class="cntcount">0</div>
  <div class="cnttext">answer</div>
</div>

<div class="qtext">
  <div class="qhead">
    {{HTML::linkRoute('question_details',
      $question->title,array($question->id,
      Str::slug($question->title)))}}
  </div>
  <div class="qinfo"">Asked by <a href="#">
    {{$asker->first_name.' '.$asker->last_name}}</a>
    around {{date('m/d/Y H:i:s',
    strtotime($question->created_at))}}</div>
  @if($tags!=null)
    <ul class="qtagul">
      @foreach($tags as $tag)
        <li>{{HTML::linkRoute('tagged',$tag->tag,
          $tag->tagFriendly)}}</li>
      @endforeach
    </ul>
```

```
            @endif
          </div>
        </div>
      @endforeach

      {{-- and lastly, the pagination --}}
      {{$questions->links()}}

    @else
      No questions found. {{HTML::linkRoute('ask',
        'Ask a question?')}}
    @endif

  @stop
```

Since we've set the relations, we can directly use `$question->users` to access the poser, or `$question->tags` to access the question's tags directly.

The method `links()` brings Laravel's built-in pagination system. The system is ready to be used with Bootstrap. Also we can alter its appearance from the `view.php` file under `app/config`.

If you followed up until here, when you navigate to your index page, after inserting some new questions, you will see a view like the following screenshot:



Now, we need to add functionality to the upvote and downvote buttons.

Adding upvote and downvote functionality

The upvote and downvote buttons will be on almost every page in our project, so adding them to the master page is a better practice instead of adding and cloning them into each template more than once.

To do this, open your `template_masterpage.php` file under `app/views`, and find the following line:

```
@yield('footer_assets')
```

Add the following code below the previous code:

```
{{-- if the user is logged in and on index or question details
  page--}}
@if(Sentry::check() && (Route::currentRouteName() ==
  'index' || Route::currentRouteName() == 'question_details'))
  <script type="text/javascript">
    $('.questions .arrowbox .like, .questions .arrowbox
    .dislike').click(function(e){
      e.preventDefault();
      var $this = $(this);
      $.get($(this).attr('href'),function($data){
        $this.parent('.arrowbox').next('.cntbox').find
          ('.cntcount').text($data);
      }).fail(function(){
        alert('An error has been occurred, please try again
        later');
      });
    });
  </script>
@endif
```

In the previous code, we check whether the user is logged in, and whether the user has navigated either to the index or the details page. Then we use JavaScript to prevent the user from clicking on the link, and we alter the click event to be an Ajax `get()` request. In the next code we will fill the vote's value with the result, which will come from the `Ajax()` request.

Now we need to write the vote update method to make it work correctly. For this, open your `QuestionsController.php` file under `app/controllers`, and add the following code:

```
/**
  * Vote AJAX Request
**/
public function getVote($direction,$id) {

  //request has to be AJAX Request
  if(Request::ajax()) {

    $question = Question::find($id);

    //if the question id is valid
    if($question) {

      //new vote count
      if($direction == 'up') {
        $newVote = $question->votes+1;
      } else {
        $newVote = $question->votes-1;
      }

      //now the update
      $update = $question->update(array(
        'votes' => $newVote
      ));

      //we return the new number
      return $newVote;
    } else {
      //question not found
      Response::make("FAIL", 400);
    }
  } else {
    return Redirect::route('index');
  }
}
```

The `getVote()` method checks whether the question is valid, and if it's valid, it increases or decreases its vote count by one. We didn't validate the parameter `$direction` here, because we've already prefiltered using regular expression at the resource that the value of `$direction` should either be `up` or `down`.

In real-world cases, you should even store the votes in a new table and check whether the users' votes are unique. You should also make sure that a user votes only once.

Now that our index page is ready and functioning, we can proceed to the next step.

Creating our questions page

In the details page, we need to show the full question to the user. Also there will be a place for the answers. To create our question page, we perform the following steps:

1. First, we need to add the details method what we've defined on our route earlier. Add the following code to your `QuesionsController.php` file under `app/controllers`:

```
/**
 * Details page
**/
public function getDetails($id,$title) {
  //First, let's try to find the question:
  $question = Question::with('users','tags')->find($id);

  if($question) {

    //We should increase the "viewed" amount
    $question->update(array(
      'viewed' => $question->viewed+1
    ));

    return View::make('qa.details')
      ->with('title',$question->title)
      ->with('question',$question);

  } else {
    return Redirect::route('index')
    ->with('error','Question not found');
  }
}
```

   We first try to fetch the question information using tags and the poser's information. If the question is found, we increase the view count by one, and we simply load the view, and add the title and the question information to the view.

2. Before displaying the view, we first need some extra routes to delete the question and reply to the post. To add these, add the following code to your `routes.php` file in the `app` folder:

```
//Reply Question:
Route::post('question/{id}/{title}',array('as'=>
  'question_reply','before'=>'csrf|user',
  'uses'=>'AnswersController@postReply'))->
  where(array('id'=>'[0-9]+','title'=>'[0-9a-zA-Z\-\_]+'));

//Admin Question Deletion
Route::get('question/delete/{id}',array('as'=>'
  delete_question','before'=>'access_check:admin',
  'uses'=>'QuestionsController@getDelete'))->
  where('id','[0-9]+');
```

3. Now that the controller method and the routes required in the view are ready, we need the view to show the data to the end user. Follow the steps and add all the code provided further, by parts, to the `details.blade.php` file under `app/views/qa`:

```
@extends('template_masterpage')

@section('content')

<h1 id="replyh">{{$question->title}}</h1>
<div class="qwrap questions">
  <div id="rcount">Viewed {{$question->viewed}}
    time{{$question->viewed>0?'s':''}}.</div>

  @if(Sentry::check())
    <div class="arrowbox">
      {{HTML::linkRoute('vote',''array('up',$question->id)
        ,array('class'=>'like', 'title'=>'Upvote'))}}
      {{HTML::linkRoute('vote','',array('down',
        $question->id),array('class'=>'dislike','title'=>
        'Downvote'))}}
    </div>
  @endif

  {{-- class will differ on the situation --}}
  @if($question->votes > 0)
    <div class="cntbox cntgreen">
  @elseif($question->votes == 0)
    <div class="cntbox">
  @else
    <div class="cntbox cntred">
```

```
@endif
    <div class="cntcount">{{$question->votes}}</div>
    <div class="cnttext">vote</div>
  </div>
```

In the first section of the view, we extend the view file to our master page `template_masterpage`. Then we start to fill the code for the section `content`. We made two links using named routes for upvoting and downvoting that will be handled using Ajax. Also, since we have different styles for each voting state (green for a positive vote and red for a negative vote), we used an `if` clause and altered the opening `<div>` tag.

4. Now add the following code to `details.blade.php`:

```
<div class="rblock">
  <div class="rbox">
    <p>{{nl2br($question->question)}}</p>
  </div>
  <div class="qinfo">Asked by <a href="#">
    {{$question->users->first_name.' '.$question->
    users->last_name}}</a> around {{date('m/d/Y
    H:i:s',strtotime($question->created_at))}}</div>

  {{--if the question has tags, show them --}}
  @if($question->tags!=null)
    <ul class="qtagul">
      @foreach($question->tags as $tag)
        <li>{{HTML::linkRoute('tagged',$tag->tag,
          $tag->tagFriendly)}}</li>
      @endforeach
    </ul>
  @endif
```

In this section, we are showing the question itself, and checking whether there are tags. If the `tags` object is not null (tags are present), we make a link with a named route for each tag, to show the questions tagged with `$tag`.

5. Now add the following code to `details.blade.php`:

```
{{-- if the user/admin is logged in, we will have a
  buttons section --}}
@if(Sentry::check())
  <div class="qwrap">
    <ul class="fastbar">
      @if(Sentry::getUser()->hasAccess('admin'))
```

```
            <li class="close">{{HTML::linkRoute(
              'delete_question','delete',$question->id)}}
            </li>
          @endif
          <li class="answer"><a href="#">answer</a></li>
        </ul>
      </div>
    @endif
  </div>
  <div id="rreplycount">{{count($question->answers)}}
    answers</div>
```

In this section, if the end user is an admin, we show buttons to answer and delete questions.

6. Now add the following code to `details.blade.php`:

```
{{-- if it's a user, we will also have the answer block
  inside our view--}}
@if(Sentry::check())
  <div class="rrepol" id="replyarea" style=
    "margin-bottom:10px">
    {{Form::open(array('route'=>array(
      'question_reply',$question->id,
      Str::slug($question->title))))}}
    <p class="minihead">Provide your Answer:</p>
    {{Form::textarea('answer',Input::old('answer'),
      array('class'=>'fullinput'))}}
    {{Form::submit('Answer the Question!')}}
    {{Form::close()}}
  </div>
@endif

</div>
@stop
```

In this section, we are adding the answering block to the question itself, benefitting from Laravel 4's built-in `Form` class. This form will only be available for logged in users (and for the admins, since they are also logged in users). We finish the section content using `@stop`.

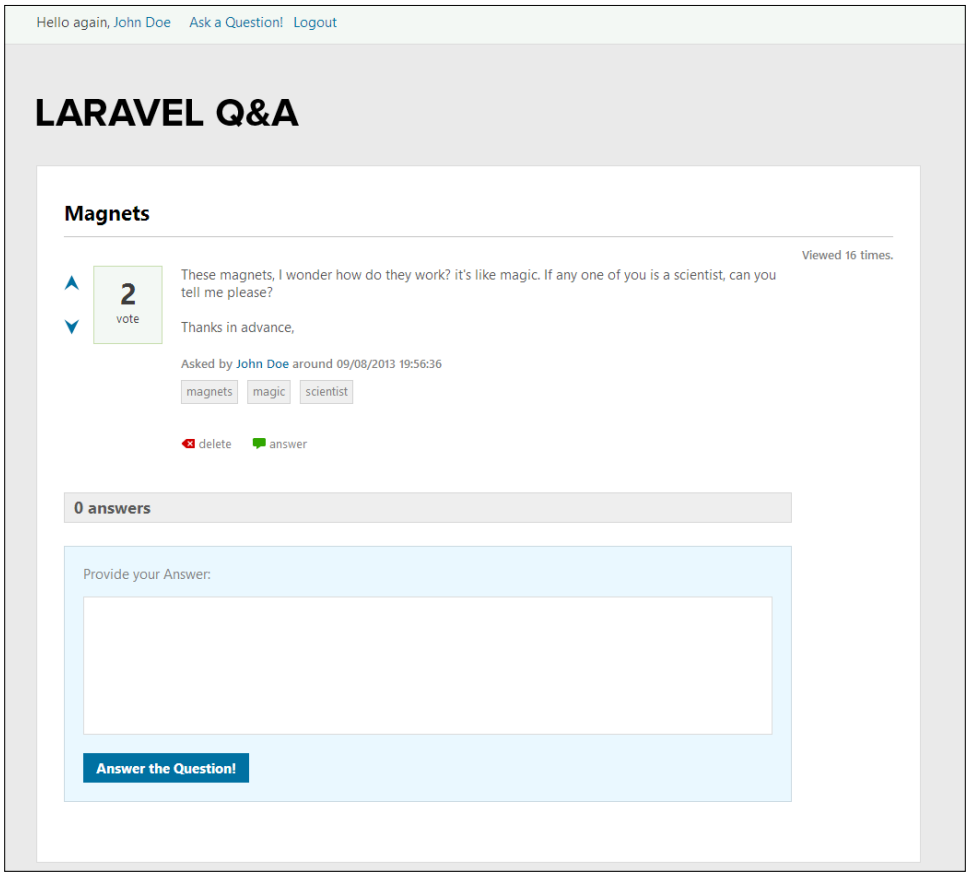
7. Now add the following code to `details.blade.php`:

```
@section('footer_assets')

  {{--If it's a user, hide the answer area and make a
    simple show/hide button --}}
```

```
    @if(Sentry::check())
      <script type="text/javascript">

      var $replyarea = $('div#replyarea');
      $replyarea.hide();

      $('li.answer a').click(function(e){
        e.preventDefault();

        if($replyarea.is(':hidden')) {
          $replyarea.fadeIn('fast');
        } else {
          $replyarea.fadeOut('fast');
        }
      });
      </script>
    @endif

    {{-- If the admin is logged in, make a confirmation to
      delete attempt --}}
    @if(Sentry::check())
      @if(Sentry::getUser()->hasAccess('admin'))
        <script type="text/javascript">
        $('li.close a').click(function(){
          return confirm('Are you sure you want to delete
            this? There is no turning back!');
        });
        </script>
      @endif
    @endif
@stop
```

In this section, we fill the `footer_assets` section to add some JavaScript to show/hide the answer field to the users, and a confirmation box is displayed to the admin before deleting the question.

If all the steps are performed, you should have a view like the following screenshot:



Lastly, we need a method to delete the question. Add the following code to your `QuestionsController.php` file under `app/controllers`:

```
/**
 * Deletes the question
**/

public function getDelete($id) {
  //First, let's try to find the question:
  $question = Question::find($id);

  if($question) {
    //We delete the question directly
    Question::delete();
```

```
        //We won't have to think about the tags and the answers,
        //because they are set as foreign key and we defined them
          cascading on deletion,
        //they will be automatically deleted

        //Let's return to the index page with a success message
        return Redirect::route('index')
          ->with('success','Question deleted successfully!');
      } else {
        return Redirect::route('index')
          ->with('error','Nothing to delete!');
      }
    }
```

Since we've set the related tables to cascade on deletion, we won't have to worry about deleting the answers and the tags while deleting a question.

Now that we are ready to post answers, we should create the answers table and process our answers.

Creating our answers table and resources

Our answers table will be very similar to the current questions table, only that it will have fewer columns. Our answers can also be voted, and an answer can be marked as the best answer either by a question's poser or an admin. To create our answers table and resources, we perform the following steps:

1. First, let's create the database table. Run the following command in the terminal:

   ```
   php artisan migrate:make create_answers_table --table=
     answers --create
   ```

2. Now, open the migration, which is created under `app/database/migrations`, and replace the `up()` function's contents with the following code:

   ```
   Schema::create('answers', function(Blueprint $table)
   {
     $table->increments('id');

     //question's id
     $table->integer('questionID')->unsigned()->default(0);
     //answerer's user id
   ```

```
    $table->integer('userID')->unsigned()->default(0);
    $table->text('answer');
    //if the question's been marked as correct
    $table->enum('correct',array('0','1'))->default(0);
    //total number of votes:
    $table->integer('votes')->default(0);
    //foreign keys
    $table->foreign('questionID')->references('id')->
      on('questions')->onDelete('cascade');
    $table->foreign('userID')->references('id')->
      on('users')->onDelete('cascade');

    $table->timestamps();
});
```

3. Now, to benefit from the Eloquent ORM and its relations, we need a model for the `answers` table. Add the following code as `Answer.php` under `app/models`:

```
<?php

class Answer extends Eloquent {

  //The relation with users
  public function users() {
    return $this->belongsTo('User','userID');
  }

  //The relation with questions
  public function questions() {
    return $this->belongsTo('Question','questionID');
  }

  //which fields can be filled
  protected $fillable = array('questionID', 'userID',
    'answer', 'correct', 'votes');

  //Answer Form Validation Rules
  public static $add_rules = array(
    'answer' => 'required|min:10'
  );

}
```

The answers are children of both users and questions, that's why in our model, we should use `belongsTo()` for users and questions to relate their tables.

4. Since a question may have more than one answer, we should also add a relation from the `questions` table to the `answers` table (to get the data about the answers to your question, all of the answers to your questions, or all of my upvoted questions' answers). To do this, open your `Question.php` file under `app/models` and add the following code:

```
public function answers() {
  return $this->hasMany('Answer','questionID');
}
```

5. Finally, we need a controller to process the requests related to answers. Run the following command in the terminal to make a controller for the answers:

```
php artisan controller:make AnswersController
```

   This command will create a file `AnswersController.php` under `app/controllers`.

Now that our answers' resource is ready, we can process the answers.

Processing the answers

In the previous section, we successfully created a question with tags, and our answers form. We now have to process the answers and add them to the database. There are some simple steps to follow for this:

1. First, we need the controller form to process the answers and add them to the table. To do this, open your freshly created `AnswersController.php` file under `app/controllers`, remove every autogenerated method inside the class, and add the following code inside the class definition:

```
/**
 * Adds a reply to the questions
**/
public function postReply($id,$title) {

  //First, let's check if the question id is valid
  $question = Question::find($id);

  //if question is found, we keep on processing
  if($question) {
```

```
    //Now let's run the form validation
    $validation = Validator::make(Input::all(),
      Answer::$add_rules);

    if($validation->passes()) {

      //Now let's create the answer
      Answer::create(array(
        'questionID' => $question->id,
        'userID' => Sentry::getUser()->id,
        'answer' => Input::get('answer')
      ));

      //Finally, we redirect the user back to the question page
with a success message
      return Redirect::route('question_details',
        array($id,$title))
        ->with('success','Answer submitted successfully!');

    } else {
      return Redirect::route('question_details',
        array($id,$title))
        ->withInput()
        ->with('error',$validation->errors()->first());
    }

  } else {
    return Redirect::route('index')
      ->with('error','Question not found');
  }

}
```

The `postReply()` method simply checks whether the question is valid, runs a form validation, adds an answer owned by the question and the user to the database, and returns the user back to the questions page.

2. Now in the questions page, we also need to include the answers and the number of answers. But before that, we need to fetch them. There are some steps to do this.

    1. First, open your `QuestionsController.php` file under `app/controllers`, and find the following line:

```
$question = Question::with('users','tags')->
  find($id);
```

Replace the previous line with the following line:

```
$question = Question::with('users','tags',
  'answers')->find($id);
```

2. Now, find the following line in the `MainController.php` file under `app/controllers`, and find this line:

```
->with('questions',Question::with('users','tags')->
  orderBy('id','desc')->paginate(2));
```

Replace the previous line with the following line:

```
->with('questions',Question::with('users', 'tags',
  'answers')->orderBy('id','desc')->paginate(2));
```

3. Now open your `index.blade.php` file under `app/views/qa`, and find the following code:

```
{{--Answer section will be filled later in
  this chapter--}}
<div class="cntbox">
  <div class="cntcount">0</div>
  <div class="cnttext">answer</div>
</div>
```

Replace the previous code with the following code:

```
<?php
//does the question have an accepted answer?
$answers = $question->answers;
$accepted = false; //default false

//We loop through each answer, and check if there
  is an accepted answer
if($question->answers!=null) {
  foreach ($answers as $answer) {
    //If an accepted answer is found, we break
      the loop
    if($answer->correct==1) {
      $accepted=true;
      break;
    }
  }
}
?>
@if($accepted)
  <div class="cntbox cntgreen">
```

```
@else
  <div class="cntbox cntred">
@endif
  <div class="cntcount">{{count($answers)}}</div>
  <div class="cnttext">answer</div>
</div>
```

In this alteration, we added a PHP code and a loop, checking each answer if it's an accepted one. And if it is, we change the `div` holder class. Also we added a feature to show the number of answers.

3. Next, we need the route resources defined to answer upvoting and downvoting and choose the best answer. Add the following code into your `routes.php` file under the `app` folder:

```
//Answer upvoting and Downvoting
Route::get('answer/vote/{direction}}/{id}',
  array('as'=>'vote_answer', 'before'=>'user',
  'uses'=>'AnswersController@getVote'))->
  where(array('direction'=>'(up|down)', 'id'=>'[0-9]+'));
```

4. Now we need to display the answers in the question details page so that the users can see the answers. To do this, open the `details.blade.php` file under `app/views/qa`, and perform the following steps:

    1. First, find the following line:

```
<div id="rreplycount">0 answers</div>
```

    Replace the previous line with the following line:

```
<div id="rreplycount">{{count($question->answers)}}
  answers</div>
```

    2. Now find the following code:

```
</div>
@stop

@section('footer_assets')
```

    Add the following code above the previous code:

```
@if(count($question->answers))
  @foreach($question->answers as $answer)

    @if($answer->correct==1)
      <div class="rrepol correct">
    @else
      <div class="rrepol">
@endif
```

```
        @if(Sentry::check())
          <div class="arrowbox">
            {{HTML::linkRoute('vote_answer','',
              array('up', $answer->id),array('class'=>
              'like', 'title'=>'Upvote'))}}
            {{HTML::linkRoute('vote_answer','',
              array('down',$answer->id),
              array('class'=>'dislike','title'=>'Downvote'
            ))}}

          </div>
        @endif

        <div class="cntbox">
          <div class="cntcount">{{$answer->votes}}</div>
          <div class="cnttext">vote</div>
        </div>

        @if($answer->correct==1)
          <div class="bestanswer">best answer</div>
        @else
          {{-- if the user is admin or the owner of the
            question, show the best answer button --}}
          @if(Sentry::check())
            @if(Sentry::getUser()->hasAccess('admin') ||
              Sentry::getUser()->id == $question->userID)
                <a class="chooseme" href="{{URL::route
                  ('choose_answer',$answer->id)}}"><div
                  class="choosebestanswer">choose</div></a>
            @endif
          @endif
        @endif
        <div class="rblock">
          <div class="rbox">
            <p>{{nl2br($answer->answer)}}</p>
          </div>
          <div class="rrepolinf">
          <p>Answered by <a href="#">{{$answer->
            users->first_name.' '.$answer->users->
            last_name}}</a> around {{date('m/d/Y H:i:s',
            strtotime($answer->created_at))}}</p>
          </div>
        </div>
      </div>
      @endforeach
    @endif
```

The current structure of answers is very close to the questions structure, which we had created earlier in this chapter. In addition, we have a button to choose the best answer, which is shown only to the poser of the question and to the admin.

3. Now, we need a confirmation button in the same view. For this, add the following code to the `footer_assets` section:

```
{{-- for admins and question owners --}}
@if(Sentry::check())
  @if(Sentry::getUser()->hasAccess('admin') ||
    Sentry::getUser()->id == $question->userID)
    <script type="text/javascript">
      $('a.chooseme').click(function(){
        return confirm('Are you sure you want to
          choose this answer as best answer?');
      });
    </script>
  @endif
@endif
```

5. Now, we need a method to increase or decrease the votes of the answers. Add the following code to your `AnswersController.php` file under `app/controllers`:

```
/**
  * Vote AJAX Request
**/
public function getVote($direction, $id) {

  //request has to be AJAX Request
  if(Request::ajax()) {
    $answer = Answer::find($id);
    //if the answer id is valid
    if($answer) {
      //new vote count
      if($direction == 'up') {
        $newVote = $answer->votes+1;
      } else {
        $newVote = $answer->votes-1;
      }

      //now the update
      $update = $answer->update(array(
        'votes' => $newVote
      ));
```

```
            //we return the new number
            return $newVote;
          } else {
            //answer not found
            Response::make("FAIL", 400);
          }
        } else {
          return Redirect::route('index');
        }
    }
```

The `getVote()` method is exactly the same as the questions voting method. The only difference here is that, instead of the question, the answers are affected.

Choosing the best answer

We need a processing method to choose a selected answer as the best answer. To choose the best answer, we perform the following steps:

1. Open your `AnswersController.php` file under `app/controllers`, and add the following code:

```
/**
  * Chooses a best answer
**/
public function getChoose($id) {

  //First, let's check if there is an answer with that given ID
  $answer = Answer::with('questions')->find($id);

  if($answer) {
    //Now we should check if the user who clicked is an
      admin or the owner of the question
    if(Sentry::getUser()->hasAccess('admin') || $answer->
      userID == Sentry::getUser()->id) {
        //First we should unmark all the answers of the
          question from correct (1) to incorrect (0)
        Answer::where('questionID',$answer->questionID)
          ->update(array(
            'correct' => 0
          ));

        //And we should mark the current answer as correct/
          best answer
      $answer->update(array(
        'correct' => 1
```

```
        ));

        //And now let's return the user back to the
          questions page
        return Redirect::route('question_details',
          array($answer->questionID, Str::slug($answer->
          questions->title)))
            ->with('success','Best answer chosen
              successfully');
      } else {
        return Redirect::route('question_details',
          array($answer->questionID, Str::slug($answer->
          questions->title)))
          ->with('error','You don\'t have access to this
            attempt!');
      }

    } else {
      return Redirect::route('index')
        ->with('error','Answer not found');
    }

}
```

In the previous code, we first check whether the answer is a valid answer. Then, we check whether the user who has clicked on the **best answer** button is either the poser of the question or the application's administrator. After that, we mark all the answers of the question as unchecked (we erase all the best answer information to the answers of the question), and mark the chosen answer as the best answer. And finally, we return the form with a success message.

2. Now, we need a method to delete the answers. For this, first we need a route. Open your `routes.php` file under `app` and add the following code:

```
//Deleting an answer
Route::get('answer/delete/{id}',array('as'=>
  'delete_answer','before'=>'user', 'uses'=>
  'AnswersController@getDelete'))->where('id','[0-9]+');
```

3. Next, find the following code in the `details.blade.php` file under `app/views/qa`:

```
<p>Answered by <a href="#">{{$answer->users->
first_name.' '.$answer->users->last_name}}</a>
around {{date('m/d/Y H:i:s',strtotime(
$answer->created_at))}}</p>
```

Add the following code below the previous code:

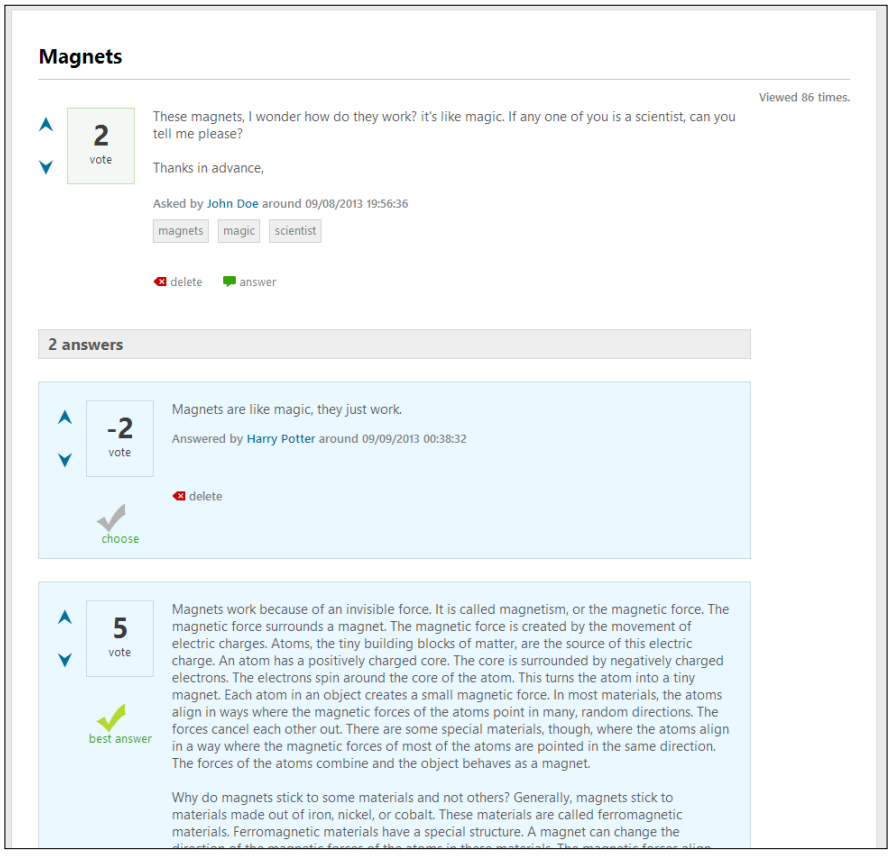
```
{{-- Only the answer's owner or the admin can delete the answer
--}}
@if(Sentry::check())
  <div class="qwrap">
    <ul class="fastbar">
      @if(Sentry::getUser()->hasAccess('admin') ||
        Sentry::getUser()->id == $answer->userID)
        <li class="close">{{HTML::linkRoute(
          'delete_answer','delete',$answer->id)}}</li>
      @endif
    </ul>
  </div>
@endif
```

4. Now, we need the controller method to delete an answer. Add the following code to the `AnswersController.php` file under `app/controllers`:

```
/**
 * Deletes an answer
**/
public function getDelete($id) {

  //First, let's check if there is an answer with
    that given ID
  $answer = Answer::with('questions')->find($id);

  if($answer) {
    //Now we should check if the user who clicked is
      an admin or the owner of the question
    if(Sentry::getUser()->hasAccess('admin') ||
      $answer->userID==Sentry::getUser()->id) {

      //Now let's delete the answer
      $delete = Answer::find($id)->delete();

      //And now let's return the user back to the
        questions page
      return Redirect::route('question_details',
        array($answer->questionID, Str::slug($answer->
        questions->title)))
        ->with('success','Answer deleted successfully');
    } else {
      return Redirect::route('question_details',
        array($answer->questionID, Str::slug($answer->
        questions->title)))
```

```
            ->with('error','You don\'t');
        }

    } else {
        return Redirect::route('index')
            ->with('error','Answer not found');
    }
}
```

If you have done everything correctly, the final version of our details page would look like the following screenshot:



Now that everything is ready to ask questions, answer, mark the best answer, and delete, only one thing is missing in our application, tag searching. As you know, we've made all the tags as links, so we should now process their routes.

Searching questions by the tags

In our main page and details page, we've given all the tags a special link. We will perform the following steps to search questions by the tags:

1. First, open your `QuestionsController.php` file under `app/controllers`, and add the following code:

```
/**
  * Shows the questions tagged with $tag friendly URL
**/
public function getTaggedWith($tag) {

  $tag = Tag::where('tagFriendly',$tag)->first();

  if($tag) {
    return View::make('qa.index')
      ->with('title','Questions Tagged with: '.$tag->tag)
      ->with('questions',$tag->questions()->
        with('users','tags','answers')->paginate(2));
  } else {
    return Redirect::route('index')
      ->with('error','Tag not found');
  }
}
```

   What this code does is, it first searches for a tag using the column `tagFriendly`, which gives a unique result. So, we can safely return the first result using `first()`. Then we check whether the tag is present in our system. If not, we return the user to the index page with an error message stating that the tag has not been found.

   If the tag is found, using the relations we've defined, we catch all the questions tagged using that tag, and we use eager loading to load the users, tags (all of the tags of the questions), and answers (although we don't show the answers on this page, we need a count of them to display it on the page). Our view would be exactly the same as the index page's view. So instead of creating a new one, we've directly used that view.

   We've kept the pagination limit to two, just to show that it works.

2. Finally, to allow JavaScript assets on the page (such as enabling Ajax upvoting and downvoting), open your `template_masterpage.php` file under `app/views`, and find the following line:

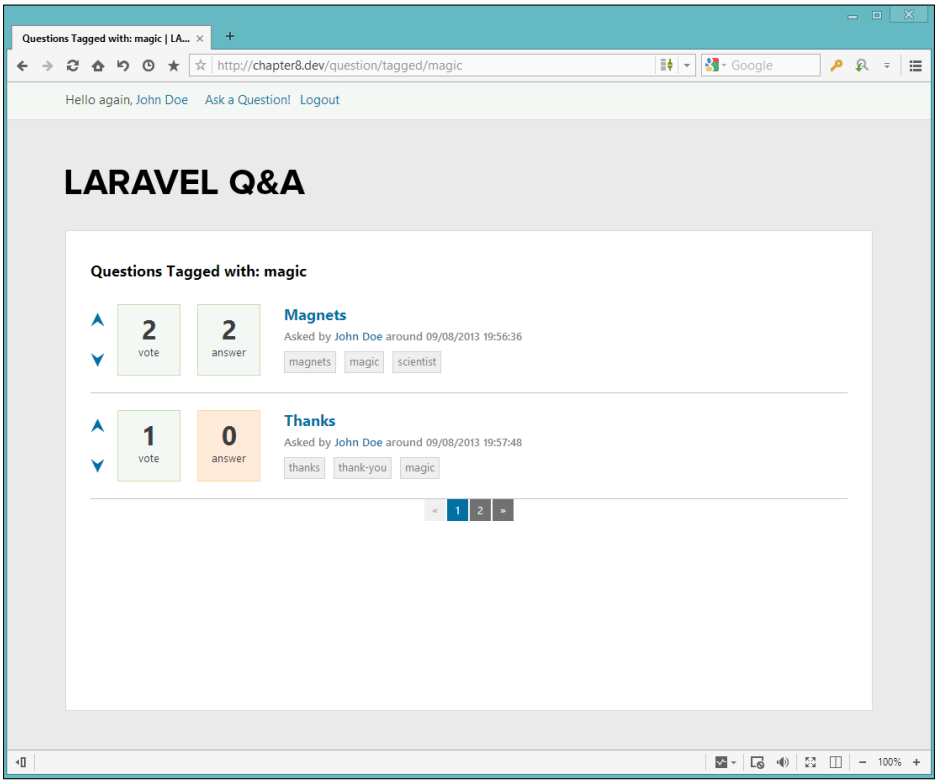
```
@if(Sentry::check() && (Route::currentRouteName() ==
  'index' || Route::currentRouteName() ==
  'question_details'))
```

   Replace the previous code with the following code:

```
@if(Sentry::check() && (Route::currentRouteName() ==
  'index' || Route::currentRouteName() == 'tagged' ||
  Route::currentRouteName() == 'question_details'))
```

This way, we allow these Ajax events even on pages having the route named `tagged`.

If you have done everything correctly, and if you click on a tag's name, a page like this will appear:



Summary

In this chapter, we've used various features of Laravel 4. We've learned to remove the public segment, to make Laravel work on some of the shared hosting solutions. We've also learned the basics of Sentry 2, a powerful authentication class. We've learned how to use many-to-many relationships and pivot tables. We've also used the belongs-to and has-any relationships using Eloquent ORM. We defined all our URLs, form actions, and links with routes using resources. So if you need to change the application's URL structure (let's say you need to change your website to German, and the German for question is frage), you only need to edit `routes.php`. So this way, you won't have to dig each file to fix the links. We've used the pagination class to navigate through records, and we've also used the Laravel Form Builder Class.

In the next chapter, we will be developing a fully featured e-commerce website using everything we've learned so far.

9

Building a RESTful API – The Movies and Actors Databases

Designing and developing a successful RESTful API is mostly very difficult. There are a lot of aspects to designing and writing a successful RESTful API; for example, securing and limiting the API. In this chapter, we'll focus on the basics of REST with coding a simple Movies and Actors API with Laravel. We'll make some JSON endpoints behind a basic authentication system, and will also learn a few Laravel 4 tricks. We'll cover the following topics in this chapter:

- Creating and migrating the users database
- Configuring the users model
- Adding sample users
- Creating and migrating the movies database
- Creating a movie model
- Adding sample movies
- Creating and migrating the actors database
- Creating an actor model
- Assigning actors to movies
- Understanding the authentication mechanism
- Querying the API

Creating and migrating the users database

We assume that you have already defined database credentials in the `database.php` file located at `app/config/`. For this application, we need a database. You can create a new database by simply running the following SQL command, or basically you can use your database administration interface such as phpMyAdmin:

```
CREATE DATABASE laravel_api
```

After successfully creating the database for the application, first we need to generate an application key for our application. As you know from the previous chapters, this is necessary for the security and authentication class of our application. To do this, first open your terminal, navigate to your project folder, and run the following command:

```
php artisian key:generate
```

If no error occurs, we should edit the authentication class' configuration file. For using Laravel's built-in authentication class, we need to edit the configuration file, `auth.php`, which is located at `app/config/`. This file contains several options for the authentication facilities. If you need to change the table name, and so on, you can perform the changes in the `auth.php` file. By default, Laravel comes with a users model; you can see the `User.php` file that is located at `app/models/`. With Laravel 4, we need to define which fields can be filled in our `User` model. Let's edit `app/models/User.php` and add the "fillable" array:

```
<?php

use Illuminate\Auth\UserInterface;
use Illuminate\Auth\Reminders\RemindableInterface;

class User extends Eloquent implements UserInterface,
RemindableInterface {

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

  /**
   * The attributes excluded from the model's JSON form.
   *
   * @var array
```

```
     */
    protected $hidden = array('password');

    // Specify which attributes should be mass-assignable
     protected $fillable = array('email', 'password');

    /**
     * Get the unique identifier for the user.
     *
     * @return mixed
     */
    public function getAuthIdentifier()
    {
      return $this->getKey();
    }

    /**
     * Get the password for the user.
     *
     * @return string
     */
    public function getAuthPassword()
    {
      return $this->password;
    }

    /**
     * Get the e-mail address where password reminders are sent.
     *
     * @return string
     */
    public function getReminderEmail()
    {
      return $this->email;
    }

}
```

Basically we need two columns for our RESTful API users, they are:

- `email`: This column stores the author's e-mail ID
- `password`: This column is for storing the author's password

Now we need several migration files to create the `users` table and add an author to our database. To create a migration file, give a command such as the following:

```
php artisan migrate:make create_users_table --table=users --create
```

Open the migration file that was created recently and located at `app/database/migrations/`. We need to edit the `up()` function as follows:

```
public function up()
{
  Schema::create('users', function(Blueprint $table)
  {
    $table->increments('id');
    $table->string('email');
    $table->string('password');
    $table->timestamps();
  });
}
```

After editing the migration file, please run the `migrate` command:

```
php artisian migrate
```

As you know, the command creates the `users` table and its columns. If no error occurs, check the `laravel_api` database for the `users` table and the columns.

Adding sample users

Now we need to create a new migration file for adding some API users to the database:

```
php artisan migrate:make add_some_users
```

Open up the migration file and edit the `up()` function as follows:

```
public function up()
  {
    User::create(array(
            'email' => 'john@gmail.com',
            'password' => Hash::make('johnspassword'),
          ));
    User::create(array(
            'email' => 'andrea@gmail.com',
            'password' => Hash::make('andreaspassword'),
          ));
  }
```

Now we have two API users for our application. The users will be accessible for querying our RESTful API.

Creating and migrating the movies database

For a simple Movies and Actors application, basically we need two tables for storing data. One of them is the `movies` table. The table will contain the name of the movie and its release year.

We need a migration file to create our `movies` table and its columns. We'll do it again with the `artisan` tool. Open your terminal, navigate to your project's folder, and run the following command:

```
php artisan migrate:make create_movies_table --table=movies --create
```

Open the migration file that was created recently and located at `app/database/migrations/`. We need to edit the `up()` function as follows:

```
public function up()
{
  Schema::create('movies', function(Blueprint $table)
  {
    $table->increments('id');
    $table->string('name');
    $table->integer('release_year');
    $table->timestamps();
  });
}
```

After editing the migration file, run the `migrate` command:

```
php artisian migrate
```

Creating a movie model

As you know, for anything related to database operations on Laravel, using models is the best practice. We will use the benefits of **Eloquent ORM**.

Save the following code in the `Movie.php` file under `app/models/`:

```
<?php
class Movie extends Eloquent {
```

```
    protected $table = 'movies';

    protected $fillable = array('name','release_year');

    public function Actors(){

          return $this-> belongsToMany('Actor' , 'pivot_table');
    }

    }
```

We have set the database table name with the protected `$table` variable. Also, we set the editable column's `$fillable` variable, and for timestamps with a `$timestamps` variable, as we've already seen and used in the previous chapters. The variables that are defined in the model are enough for using Laravel's Eloquent ORM. We'll cover the public `Actor()` function in the *Assigning actors to movies* section in this chapter.

Our movie model is ready: now we need an actor model and its corresponding table.

Adding sample movies

Now we need to create a new migration file for adding some movies to the database. Actually, you can also use the database seeder for seeding the database. Here, we will use migration files for seeding the database. You can check out the seeders at:

`http://laravel.com/docs/migrations#database-seeding`

Run the following `migrate` command:

```
php artisan migrate:make add_some_movies
```

Open up the migration file and edit the `up()` function as follows:

```
    public function up()
    {
      Movie::create(array(
              'name' => 'Annie Hall',
          'release_year' => '1977'
            ));

      Movie::create(array(
              'name' => ' Manhattan ',
          'release_year' => '1978'
            ));

    Movie::create(array(
```

```
            'name' => 'The Shining',
        'release_year' => '1980'
          ));
    }
```

Creating and migrating the actors database

We need to create an `actors` table that will contain the names of the actors of the movies. We need a migration file to create our `movies` table and columns. We'll do it again with the `artisan` tool. Let's open up our terminal, navigate to our project folder, and run the following command:

```
php artisan migrate:make create_actors_table --table=actors -create
```

Open the migration file that was created recently and located at `app/database/migrations/`. We need to edit the `up()` function as follows:

```
public function up()
{
  Schema::create('actors', function(Blueprint $table)
  {
    $table->increments('id');
    $table->string('name');
    $table->timestamps();
  });
}
```

After editing the migration file, run the `migrate` command as follows:

```
php artisian migrate
```

Creating an actor model

For creating the actor model, save the following code as `Movies.php` under `app/models/`:

```
<?php
class Actor extends Eloquent {

protected $table = 'actors';

protected $fillable = array('name');
```

```
public function Movies(){

      return $this-> belongsToMany('Movies', 'pivot_table');
}

}
```

Assigning actors to movies

As you know, we used the `belongsToMany` relation between the actors and movie models. This is because an actor has probably acted in many movies. A movie also would probably have many actors.

As you will see, in the previous sections of this chapter, we used a pivot table named `pivot_table`. We can also create the pivot table with the `artisan` tool. Let's create it:

```
php artisan migrate:make create_pivot_table --table=pivot_table
--create
```

Open the migration file that was created recently and located at `app/database/migrations/`. We need to edit the `up()` function as follows:

```
  public function up()
  {
    Schema::create('pivot_table', function(Blueprint $table)
    {
      $table->increments('id');
      $table->integer('movies_id');
      $table->integer('actors_id');
      $table->timestamps();
    });
}
```

After editing the migration file, run the `migrate` command:

```
php artisian migrate
```

Now we need to create a new migration file for adding some actors to the database:

```
php artisan migrate:make add_some_actors
```

Open up the migration file and edit the `up()` function as follows:

```
  public function up()
  {
    $woody = Actor::create(array(
```

```
            'name' => 'Woody Allen'
          ));

    $woody->Movies()->attach(array('1','2'));

    $diane = Actor::create(array(
            'name' => 'Diane Keaton'
          ));

$diane->Movies()->attach(array('1','2'));

$jack = Actor::create(array(
            'name' => 'Jack Nicholson'
        ));

$jack->Movies()->attach(3);

}
```

Let's grab the migration file. When we attach `users` to `movies`, we've to use the movie IDs shown as follows:

```
$voody = Actor::create(array(
            'name' => 'Woody Allen'
          ));

$voody->Movies()->attach(array('1','2'));
```

This means *Woody Allen* has played a role in two films, and the ID of these movies are `1` and `2`. Also, *Diane Keaton* has played a role in those two movies. But *Jack Nicholson* has played a role in *The Shining* and the ID of the film is `3`. As we have already elaborated on the Eloquent ORM relations in *Chapter 8*, *Building a Q&A Web Application*, our relation type is the **Eloquent belongsToMany** relation.

Understanding the authentication mechanism

Like many other APIs, our API system is authentication based. As you may remember from the previous chapters, Laravel comes with an authentication mechanism. In this section, we'll use the pattern-based route filtering feature of Laravel for securing and limiting our API. First, we need to edit the `auth.basic` filter for our application.

Open the route filter configuration file that is located at `app/filters.php` and edit the `auth.basic` filter as follows:

```
Route::filter('auth.basic', function()
{
  return Auth::basic('email');
});
```

The API users should be sending their e-mail IDs and passwords, along with their requests, to our application. Because of the request, we edit the filter. An API request will be as follows:

```
curl -i –user andrea@gmail.com:andreaspassword localhost/api/
getactorinfo/Woody%20Allen
```

Now, we need to apply a filter on our routes. Open the route filter configuration file that is located at `app/routes.php` and add the following code:

```
Route::when('*', 'auth.basic');
```

This code indicates that our application needs authentication for every request on it. Now we need to write our routes. Add the following lines to `app/routes.php`:

```
Route::get('api/getactorinfo/{actorname}', array('uses' =>
'ActorController@getActorInfo'));
Route::get('api/getmovieinfo/{moviename}', array('uses' =>
'MovieController@getMovieInfo'));
Route::put('api/addactor/{actorname}', array('uses' =>
'ActorController@putActor'));
Route::put('api/addmovie/{moviename}/{movieyear}', array('uses' =>
'MovieController@putMovie'));
Route::delete('api/deleteactor/{id}', array('uses' =>
'ActorController@deleteActor'));
Route::delete('api/deletemovie/{id}', array('uses' =>
'MovieController@deleteMovie'));
```

Querying the API

We need two controller files for our RESTful route functions. Let's create two controller files under `app/controllers/`. The files should be named `MovieController.php` and `ActorController.php`.

Getting movie/actor information from the API

First, we need the `getActorInfo()` and `getMovieInfo()` functions for getting actor and movie information from the database. Open the `ActorController.php` file located at `app/controllers/` and write the following code:

```
<?php

class ActorController extends BaseController {
public function getActorInfo($actorname){

$actor = Actor::where('name', 'like', '%'.$actorname.'%')->first();
if($actor){

$actorInfo = array('error'=>false,'Actor Name'=>$actor->name,'Actor
ID'=>$actor->id);
$actormovies = json_decode($actor->Movies);
foreach ($actormovies as $movie) {
$movielist[] = array("Movie Name"=>$movie->name, "Release
Year"=>$movie->release_year);
}
$movielist =array('Movies'=>$movielist);
return Response::json(array_merge($actorInfo,$movielist));

}
else{

return Response::json(array(
'error'=>true,
'description'=>'We could not find any actor in database like
:'.$actorname
));
}
}
}
```

Next, open the `MovieController.php` file located at `app/controllers/` and write the following code:

```
<?php

class MovieController extends BaseController {
```

```
public function getMovieInfo($moviename){
$movie = Movie::where('name', 'like', '%'.$moviename.'%')->first();
if($movie){

$movieInfo = array('error'=>false,'Movie Name'=>$movie->name,'Release
Year'=>$movie->release_year,'Movie ID'=>$movie->id);
$movieactors = json_decode($movie->Actors);
foreach ($movieactors as $actor) {
$actorlist[] = array("Actor"=>$actor->name);
}
$actorlist =array('Actors'=>$actorlist);
return Response::json(array_merge($movieInfo,$actorlist));

}
else{

return Response::json(array(
'error'=>true,
'description'=>'We could not find any movie in database like
:'.$moviename
));
}
}
}
```

The functions `getActorInfo()` and `getMovieInfo()` basically search the database for the movie/actor name with the given text. If such a movie or actor is found, it is returned in the JSON format. So, for getting actor information from the API, our users can make a request as follows:

```
curl -i --user andrea@gmail.com:andreaspassword localhost/api/
getactorinfo/Woody
```

The response for the actor information request will be as follows:

```
{
   "error":false,
   "Actor Name":"Woody Allen",
   "Actor ID":1,
   "Movies":[
      {
         "Movie Name":"AnnieHall",
         "Release Year":1977
      },
```

```
        {
           "Movie Name":"Manhattan",
           "Release Year":1978
        }
     ]
  }
```

The request for any movie would be similar to this:

```
curl -i --user andrea@gmail.com:andreaspassword localhost/api/
getmovieinfo/Manhattan
```

The response for the movie information request will be as follows:

```
  {
     "error":false,
     "Movie Name":"Manhattan",
     "Release Year":1978,
     "Movie ID":2,
     "Actors":[
        {
           "Actor":"Woody Allen"
        },
        {
           "Actor":"Diane Keaton"
        }
     ]
  }
```

If any user requests movie information from an API that doesn't exist in the database, the response will look like this:

```
  {
     "error":true,
     "description":"We could not find any movie in database like
  :Terminator"
  }
```

Also, a similar response will be for an actor that doesn't exist in the database:

```
  {
     "error":true,
     "description":"We could not find any actor in database like :Al
  Pacino"
  }
```

Sending new movies/actors to the API's database

We need the `putActor()` and `putMovie()` functions for allowing users to add new actors/movies to our database.

Open the `ActorController.php` file located at `app/controllers/` and add the following function:

```
public function putActor($actorname)
{

$actor = Actor::where('name', '=', $actorname)->first();
if(!$actor){

$the_actor = Actor::create(array('name'=>$actorname));

return Response::json(array(
'error'=>false,
'description'=>'The actor successfully saved. The ID number of Actor
is : '.$the_actor->id
));

}
else{

return Response::json(array(
'error'=>true,
'description'=>'We have already in database : '.$actorname.'. The ID
number of Actor is : '.$actor->id
));
}
}
```

Now open the `MovieController.php` file located at `app/controllers/` and add the following function:

```
public function putMovie($moviename,$movieyear)
{

$movie = Movie::where('name', '=', $moviename)->first();
if(!$movie){
```

```
$the_movie = Movie::create(array('name'=>$moviename,'release_
year'=>$movieyear));

return Response::json(array(
'error'=>false,
'description'=>'The movie successfully saved. The ID number of Movie
is : '.$the_movie->id
));

}
else{

return Response::json(array(
'error'=>true,
'description'=>'We have already in database : '.$moviename.'. The ID
number of Movie is : '.$movie->id
));
}
}
```

The functions `putActor()` and `putMovie()` basically search the database for movies/actors names with the given text. If there is a movie or actor found, the functions return its ID in the JSON format, else it creates the new actor/movie and responds with the new record ID. So, for creating a new actor in the API database, our users can make a request such as the following:

```
curl -i -X PUT --user andrea@gmail.com:andreaspassword localhost/api/
addactor/Al%20Pacino
```

The response for the movie information request will be as follows:

```
{
   "error":false,
   "description":"The actor successfully saved. The ID number of Actor
is : 4"
}
```

If any API user tries to add the existing actor, the API will respond as follows:

```
{
   "error":true,
   "description":"We have already in database : Al Pacino. The ID
number of Actor is : 4"
}
```

Also, the response for creating a new movie in the API database should be as follows:

```
curl -i –X PUT –-user andrea@gmail.com:andreaspassword localhost/api/
addmovie/The%20Terminator/1984
```

The response for the request will be as follows:

```
{
   "error":false,
   "description":"The movie successfully saved. The ID number of Movie
is : 4"
}
```

If any API user tries to add the existing actor, the API will respond as follows:

```
{
   "error":true,
   "description":"We have already in database : The Terminator. The ID
number of Movie is : 4"
}
```

Deleting movies/actors from the API

Now we need the `deleteActor()` and `deleteMovie()` functions for allowing users to add new actors/movies to our database.

Open the `ActorController.php` file under `app/controllers/` and add the following function:

```
public function deleteActor($id)
{

$actor = Actor::find($id);
if($actor){

$actor->delete();

return Response::json(array(
'error'=>false,
'description'=>'The actor successfully deleted : '.$actor->name
));

}
else{
```

```
return Response::json(array(
'error'=>true,
'description'=>'We could not find any actor in database with ID number
:'.$id
));
}
}
```

After adding the function, the content in `ActorController.php` located at `app/controllers/`, should look like the following:

```
<?php
class ActorController extends BaseController
{
    public function getActorInfo($actorname)
    {
        $actor = Actor::where('name', 'like', '%' . $actorname .
'%')->first();
        if ($actor)
        {
            $actorInfo   = array(
                'error' => false,
                'Actor Name' => $actor->name,
                'Actor ID' => $actor->id
            );
            $actormovies = json_decode($actor->Movies);
            foreach ($actormovies as $movie)
            {
                $movielist[] = array(
                    "Movie Name" => $movie->name,
                    "Release Year" => $movie->release_year
                );
            }
            $movielist = array(
                'Movies' => $movielist
            );
            return Response::json(array_merge($actorInfo,
$movielist));
        }
        else
        {
            return Response::json(array(
                'error' => true,
                'description' => 'We could not find any actor in
database like :' . $actorname
```

```
            ));
        }
    }
    public function putActor($actorname)
    {
        $actor = Actor::where('name', '=', $actorname)->first();
        if (!$actor)
        {
            $the_actor = Actor::create(array(
                'name' => $actorname
            ));
            return Response::json(array(
                'error' => false,
                'description' => 'The actor successfully saved. The ID
number of Actor is : ' . $the_actor->id
            ));
        }
        else
        {
            return Response::json(array(
                'error' => true,
                'description' => 'We have already in database : ' .
$actorname . '. The ID number of Actor is : ' . $actor->id
            ));
        }
    }
    public function deleteActor($id)
    {
        $actor = Actor::find($id);
        if ($actor)
        {
            $actor->delete();
            return Response::json(array(
                'error' => false,
                'description' => 'The actor successfully deleted : ' .
$actor->name
            ));
        }
        else
        {
            return Response::json(array(
                'error' => true,
                'description' => 'We could not find any actor in
database with ID number :' . $id
```

```
            ));
        }
    }
}
```

Now we need a similar function for `MovieController`. Open the `MovieController.php` file under `app/controllers/` and add the following function:

```
public function deleteMovie($id)
{

$movie = Movie::find($id);
if($movie){

$movie->delete();

return Response::json(array(
'error'=>false,
'description'=>'The movie successfully deleted : '.$movie->name
));

}
else{

return Response::json(array(
'error'=>true,
'description'=>'We could not find any movie in database with ID number
:'.$id
));
}
}
```

After adding the function, the content under `ActorController.php` located at `app/controllers/` should look like the following:

```
<?php
class  extends BaseController
{
    public function getMovieInfo($moviename)
    {
        $movie = Movie::where('name', 'like', '%' . $moviename .
'%')->first();
        if ($movie)
        {
            $movieInfo   = array(
                'error' => false,
```

```
                'Movie Name' => $movie->name,
                'Release Year' => $movie->release_year,
                'Movie ID' => $movie->id
            );
            $movieactors = json_decode($movie->Actors);
            foreach ($movieactors as $actor)
            {
                $actorlist[] = array(
                    "Actor" => $actor->name
                );
            }
            $actorlist = array(
                'Actors' => $actorlist
            );
            return Response::json(array_merge($movieInfo,
$actorlist));
        }
        else
        {
            return Response::json(array(
                'error' => true,
                'description' => 'We could not find any movie in
database like :' . $moviename
            ));
        }
    }
    public function putMovie($moviename, $movieyear)
    {
        $movie = Movie::where('name', '=', $moviename)->first();
        if (!$movie)
        {
            $the_movie = Movie::create(array(
                'name' => $moviename,
                'release_year' => $movieyear
            ));
            return Response::json(array(
                'error' => false,
                'description' => 'The movie successfully saved. The ID
number of Movie is : ' . $the_movie->id
            ));
        }
        else
        {
            return Response::json(array(
```

```
                'error' => true,
                'description' => 'We have already in database : ' .
$moviename . '. The ID number of Movie is : ' . $movie->id
            ));
        }
    }
    public function deleteMovie($id)
    {
        $movie = Movie::find($id);
        if ($movie)
        {
            $movie->delete();
            return Response::json(array(
                'error' => false,
                'description' => 'The movie successfully deleted : ' .
$movie->name
            ));
        }
        else
        {
            return Response::json(array(
                'error' => true,
                'description' => 'We could not find any movie in
database with ID number :' . $id
            ));
        }
    }
}
```

The functions `deleteActor()` and `deleteMovie()` basically search the database for a movie/actor with the given ID. If there is a movie or an actor, the API deletes the actor/movie and returns the status in the JSON format. So, for deleting an actor from the API, our users can make a request as follows:

```
curl -I -X DELETE --user andrea@gmail.com:andreaspassword localhost/api/
deleteactor/4
```

The response for the request will be as follows:

```
{
   "error":false,
   "description":"The actor successfully deleted : Al Pacino"
}
```

Also, the response for deleting a movie from the API database should be as follows:

```
curl –I –X DELETE --user andrea@gmail.com:andreaspassword localhost/api/
deletemovie/4
```

The response for the request will be as follows:

```
{
   "error":false,
   "description":"The movie successfully deleted : The Terminator"
}
```

If any API user tries to delete a movie/actor from the API database that doesn't exist, the API will respond as follows:

```
{
   "error":true,
   "description":"We could not find any movie in database with ID
number :17"
}
```

For deleting an actor that doesn't exist, the response will be as follows:

```
{
   "error":true,
   "description":"We could not find any actor in database with ID
number :58"
}
```

Summary

In this chapter, we've focused on the basics of REST with coding a simple Movies and Actors API with Laravel. We've made some JSON endpoints behind a basic authentication system, and learned a few Laravel 4 tricks while the chapter uses something like a pattern-based route filtering. As you saw, developing and securing a RESTful application is very easy with Laravel. In the next chapter, we'll cover more effective methods in Laravel while coding a simple e-commerce application.

10
Building an E-Commerce Website

In this chapter, we'll code a simple book store example using Laravel. We'll also cover Laravel's built-in authentication, named routes, and database seeding. We'll also elaborate some rapid development methods that come with Laravel such as creating route URLs. Also, we'll working with a new relation type called *belongs to many*. We'll cover pivot tables as well. Our e-commerce application will be a simple book store. This application will have order, administration, and cart features. We will cover the following topics:

- Building an authorization system
- Creating and migrating authors, books, carts, and orders tables
- Creating template files
- Listing books
- Building a shopping cart
- Taking orders
- Listing orders

Building an authorization system

We assume that you have already defined the database credentials in the `database.php` file located at `app/config`. To create our e-commerce application, we need a database. You can create and simply run the following SQL command or basically you can use a database administration interface such as phpMyAdmin:

```
CREATE DATABASE laravel_store
```

Creating and migrating the members' database

Contrary to most of the PHP frameworks, Laravel has a basic and customizable authentication mechanism. The authentication class is very helpful for rapidly developing applications. First, we need a secret key for our application. As we mentioned in previous chapters, the application's secret key is very important for our application's security because all the data is hashed salting this key. The artisan can generate this key for us with a single command line:

```
php artisan key:generate
```

If no error occurs, you will see a message that tells you the key is generated successfully. After key generation, if you've visited your project URL before you face problems with opening your Laravel application, simply clear your browser's cache and try again. Next, we should edit the authentication class's configuration file. To use Laravel's built-in authentication class, we need to edit the configuration file, which is located at `app/config/auth.php`. This file contains several options for the authentication facilities. If you need a change in the table name, and so on, you can make the changes under this file. By default, Laravel comes with a `User` model. You can see the file, which is located at `app/models/User.php`. With Laravel 4, we need to define which fields are fillable in our `User` model. Let's edit `User.php` located at `app/models/` and add the `fillable` array:

```
<?php

use Illuminate\Auth\UserInterface;
use Illuminate\Auth\Reminders\RemindableInterface;

class User extends Eloquent implements UserInterface,
RemindableInterface {

  protected $table = 'users';

  /**
   * The attributes excluded from the model's JSON form.
   *
   * @var array
   */
  protected $hidden = array('password');

  //Add to the "fillable" array
   protected $fillable = array('email', 'password', 'name', 'admin');
```

```
    /**
     * Get the unique identifier for the user.
     *
     * @return mixed
     */
    public function getAuthIdentifier()
    {
      return $this->getKey();
    }

    /**
     * Get the password for the user.
     *
     * @return string
     */
    public function getAuthPassword()
    {
      return $this->password;
    }

    /**
     * Get the e-mail address where password reminders are sent.
     *
     * @return string
     */
    public function getReminderEmail()
    {
      return $this->email;
    }

  }
```

Basically we need four columns for our members. These are:

- `email`: This is the column for storing a member's e-mails
- `password`: This is the column for storing a member's password
- `name`: This is the column for storing a member's name and surname
- `admin`: This is the column for flagging store admin

Now we need several migration files to create the `users` table and add a member to our database. To create a migration file, give a command as follows:

```
php artisan migrate:make create_users_table --table=users --create
```

Open the migration file, which was created recently and located at `app/database/migrations/`. We need to edit the `up()` function, as shown in the following code snippet:

```
public function up()
{
  Schema::create('users', function(Blueprint $table)
  {
    $table->increments('id');
    $table->string('email');
    $table->string('password');
    $table->string('name');
    $table->integer('admin');
    $table->timestamps();
  });
}
```

After editing the `migration` file, run the `migrate` command:

```
php artisan migrate
```

Now we need to create a database seeder file to add some users to the database. Database seeding is another highly recommended way to add data to your application database. The database seeder files are located at `app/database/seeds`. Let's create our first seeder file under the `UsersTableSeeder.php` directory.

We can create both the seeder file and the seeder class with any name. But it is highly recommended for the seeder file and class name that the table name should follow camel case, for example, `TableSeeder`. Following the world-wide programming standards will improve the quality of your code.

The content of `UsersTableSeeder.php` should look like the following:

```
<?php
Class UsersTableSeeder extends Seeder {

    public function run()
    {
  DB::table('users')->delete();

User::create(array(
            'email' => 'member@email.com',
            'password' => Hash::make('password'),
            'name' => 'John Doe',
```

```
            'admin'=>0
        ));

    User::create(array(
            'email' => 'admin@store.com',
            'password' => Hash::make('adminpassword'),
            'name' => 'Jeniffer Taylor',
            'admin'=>1
        ));

    }

}
```

To apply seeding, first we need to call the `Seeder` class. Let's open the `DatabaseSeeder.php` file located at `app/database/seeds` and edit the file, as shown in the following code snippet:

```
<?php
class DatabaseSeeder extends Seeder {

  /**
   * Run the database seeds.
   *
   * @return void
   */
  public function run()
  {
    Eloquent::unguard();

    $this->call('UsersTableSeeder');
    $this->command->info('Users table seeded!');
  }

}
```

It is very important to securely store your users' passwords and their critical data. Do not forget that if you change the application key, all the existing hashed records will be unusable because the `Hash` class uses the application key as the salting key, when validating and storing given data.

Creating and migrating the authors' database

We need an Author model for storing the book authors. It will be a very simple structure. Let's create the `Author.php` file under `app/models` and add the following code:

```
<?php
Class Author extends Eloquent {

protected $table = 'authors';

protected $fillable = array('name','surname');

}
```

Now we need several migration files to create the `authors` table and add some authors to our database. To create a migration file, give a command as follows:

```
php artisan migrate:make create_authors_table --table=authors --create
```

Open the migration file that was created recently and located at `app/database/migrations/`. We need to edit the `up()` function as follows:

```
public function up()
{
  Schema::create('authors', function(Blueprint $table)
  {
    $table->increments('id');
    $table->string('name');
    $table->string('surname');
    $table->timestamps();
  });
}
```

After editing the migration file, run the `migrate` command:

```
php artisan migrate
```

As you know, the command creates the `authors` table and its columns. If no error occurs, check the `laravel_store` database for the `authors` table and the columns.

Adding authors to the database

Now we need to create a database seeder file to add some authors to the database. Let's create our first seeder file under `app/database/seeds/AuthorsTableSeeder.php`.

The content in `AuthorsTableSeeder.php` should look like the following:

```
<?php
Class AuthorsTableSeeder extends Seeder {

    public function run()
    {
DB::table('authors')->delete();

        Author::create(array(
            'name' => 'Lauren',
            'surname'=>'Oliver'
        ));

        Author::create(array(
            'name' => 'Stephenie',
            'surname'=>'Meyer'
        ));

        Author::create(array(
            'name' => 'Dan',
            'surname'=>'Brown'
        ));

    }

}
```

To apply seeding, first we need to call the `Seeder` class. Let's open the file `DatabaseSeeder.php` located at `app/database/seeds/`and edit the file as shown in the following code snippet:

```
<?php
class DatabaseSeeder extends Seeder {

  /**
   * Run the database seeds.
   *
   * @return void
   */
```

```
    public function run()
    {
      Eloquent::unguard();
      $this->call('UsersTableSeeder');
      $this->command->info('Users table seeded!');
      $this->call('AuthorsTableSeeder');
      $this->command->info('Authors table seeded!');
    }

}
```

We need to seed our database with the following `artisan` command:

```
php artisan db:seed
```

When you want to rollback and re-run all migrations, you can use the following command:

```
php artisan migrate:refresh --seed
```

Creating and migrating the books database

We need a `Book` model to store the author's books. Let's create the `Book.php` file under `app/models/` and add the following code:

```
<?php
Class Book extends Eloquent {

protected $table = 'books';

protected $fillable = array('title','isbn','cover','price','author_
id');

public function Author(){

return $this->belongsTo('Author');

}

}
```

Let's explain the role of the `author_id` column and the `Author()` function. As you know from previous chapters, `Eloquent` has several functions for different kinds of database relations. The `author_id` will store the ID of the authors. The `Author()` function is used to fetch names and surnames of authors from the `authors` table.

Adding books to the database

Now we need to create a database seeder file to add some books to the database. Let's create the first seeder file under `app/database/seeds/BooksTableSeeder.php`.

The content in `BooksTableSeeder.php` should look like the following:

```
<?php
Class BooksTableSeeder extends Seeder {

  public function run()
  {
  DB::table('books')->delete();

  Book::create(array(
    'title'=>'Requiem',
    'isbn'=>'9780062014535',
    'price'=>'13.40',
    'cover'=>'requiem.jpg',
    'author_id'=>1
   ));
  Book::create(array(
    'title'=>'Twilight',
    'isbn'=>'9780316015844',
    'price'=>'15.40',
    'cover'=>'twilight.jpg',
    'author_id'=>2
  ));
  Book::create(array(
    'title'=>'Deception Point',
    'isbn'=>'9780671027384',
    'price'=>'16.40',
    'cover'=>'deception.jpg',
    'author_id'=>3
  ));

  }

}
```

To apply seeding, first we need to call the seeder class. Let's open the `DatabaseSeeder.php` file located at `app/database/seeds` and edit the file, as shown in the following code snippet:

```
<?php
class DatabaseSeeder extends Seeder {

  /**
   * Run the database seeds.
   *
   * @return void
   */
  public function run()
  {
    Eloquent::unguard();
    $this->call('UsersTableSeeder');
    $this->command->info('Users table seeded!');
    $this->call('AuthorsTableSeeder');
    $this->command->info('Authors table seeded!');
    $this->call('BooksTableSeeder');
    $this->command->info('Books table seeded!');
  }

}
```

Now, we need to seed our database with the following `artisan` command:

```
php artisan db:seed
```

Creating and migrating the carts database

As you know, all e-commerce applications should have a cart. In this application, we'll have a cart too. We'll design a member-based cart, which means we can store and show each visitor their carts and cart items. So, we need a `Cart` model to store the cart items. It will be a very simple structure. Let's create the `Cart.php` file under `app/models` and add following code:

```
<?php
Class Cart extends Eloquent {

protected $table = 'carts';

protected $fillable = array('member_id','book_id','amount','total');
```

```
public function Books(){

return $this->belongsTo('Book','book_id');

}

}
```

Now we need a migration file to create the `carts` table. To create a migration file, give a command such as the following:

```
php artisan migrate:make create_carts_table --table=carts --create
```

Open the migration file, which was created recently and located at `app/database/migrations/`. We need to edit the `up()` function as shown in the following code snippet:

```
public function up()
{
  Schema::create('carts', function(Blueprint $table)
  {
    $table->increments('id');
    $table->integer('member_id');
    $table->integer('book_id');
    $table->integer('amount');
    $table->decimal('total', 10, 2);
    $table->timestamps();
  });
}
```

To apply migration, we need to migrate with the following `artisan` command:

```
php artisan migrate
```

Creating and migrating the orders database

To store members' orders, we need two tables. The first of them is the `orders` table, which will store shipping details, member ID, and the total value of the order. The second one is the `order_books` table. This table will store orders' books and will be our pivot table. In this model, we'll use the `belongsToMany()` relation. This is because an order can have many books.

So, first we need an `Order` model to store the book orders. Let's create the `Order.php` file under `app/models` and add the following code:

```
<?php
Class Order extends Eloquent {

protected $table = 'orders';

protected $fillable = array('member_id','address','total');

public function orderItems()
    {
        return $this->belongsToMany('Book')
->withPivot('amount','total');
    }

}
```

As you can see in the code, we've used a new option named `withPivot()` with the `belongsToMany()` function. With the `withPivot()` function, we can fetch extra fields from our pivot table. Normally, without the function, the relational query accesses from the pivot table with just the `id` object of related rows. This is necessary for our application because of price changes. Thus, previous orders, which were possibly done before any price change, are not affected.

Now we need a migration file to create the `carts` table. To create a `migration` file, give a command such as the following:

```
php artisan migrate:make create_orders_table --table=orders --create
```

Open the migration file, which was created recently and located at `app/database/migrations`. We need to edit the `up()` function as follows:

```
public function up()
{
  Schema::create('orders', function(Blueprint $table)
  {
    $table->increments('id');
    $table->integer('member_id');
    $table->text('address');
    $table->decimal('total', 10, 2);
    $table->timestamps();
  });
}
```

To apply migration, we need to migrate with the following `artisan` command:

```
php artisan migrate
```

Let's create our `pivot` table. We need a migration file to create the `order_books` table. To create a migration file, give a command such as the following:

```
php artisan migrate:make create_order_books_table --table=order_books
--create
```

Open the migration file, which was created recently and located at `app/database/migrations`. We need to edit the `up()` function as follows:

```
    public function up()
    {
      Schema::create('order_books', function(Blueprint $table)
      {
        $table->increments('id');
        $table->integer('order_id');
        $table->integer('book_id');
        $table->integer('amount');
  $table->decimal('price', 10, 2);
        $table->decimal('total', 10, 2);
      });
  }
```

To apply migration, we need to migrate with the following `artisan` command:

```
php artisan migrate
```

Our database design and models are finished. Now we need to code controllers and our application's front pages.

Listing books

First, we need to list our products. To do that, we need to create a controller, which is named `BookController`. Let's create a file under `app/controllers/` and save it with the name `BookController.php`. The controller code should look like the following:

```
<?php
class BookController extends BaseController{

  public function getIndex()
  {
```

```
        $books = Book::all();

        return View::make('book_list')->with('books',$books);

    }
}
```

The code simply fetches all the books from our `books` table and passes the data `book_list.blade.php` template with the `$books` variable. So we need to create a template file under `app/controllers/`, which is named as `book_list.blade.php`. Before doing this we need a layout page for our templates. Working with layout files is very helpful to manage html code. So first, we need a template file under `app/controllers/`, which is named `main_layout.blade.php`. The code should look like the following:

```
<!DOCTYPE html>
<html>
<head>
  <title>Awesome Book Store</title>
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <!-- Bootstrap -->
  <link href="//netdna.bootstrapcdn.com/twitter-bootstrap/2.3.1/css/
bootstrap.min.css" rel="stylesheet">
</head>
<body>
<div class="navbar navbar-inverse nav">
    <div class="navbar-inner">
        <div class="container">
            <a class="btn btn-navbar" data-toggle="collapse" data-
target=".nav-collapse">
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
            </a>
            <a class="brand" href="/">Awesome Book Store</a>
            <div class="nav-collapse collapse">
              <ul class="nav">
                  <li class="divider-vertical"></li>
                  <li><a href="/"><i class="icon-home icon-white"></i>
Book List</a></li>
              </ul>
              <div class="pull-right">
                <ul class="nav pull-right">
                @if(!Auth::check())
```

```
                <ul class="nav pull-right">
                  <li class="divider-vertical"></li>
                  <li class="dropdown">
                    <a class="dropdown-toggle" href="#" data-
toggle="dropdown">Sign In <strong class="caret"></strong></a>
                    <div class="dropdown-menu" style="padding: 15px;
padding-bottom: 0px;">
                      <p>Please Login</a>
                        <form action="/user/login" method="post"
accept-charset="UTF-8">
                          <input id="email" style="margin-bottom:
15px;" type="text" name="email" size="30" placeholder="email" />
                          <input id="password" style="margin-bottom:
15px;" type="password" name="password" size="30" />
                          <input class="btn btn-info" style="clear:
left; width: 100%; height: 32px; font-size: 13px;" type="submit"
name="commit" value="Sign In" />
                        </form>
                      </div>
                    </li>
                  </ul>
                @else
                <li><a href="/cart"><i class="icon-shopping-cart icon-
white"></i> Your Cart</a></li>
                  <li class="dropdown"><a href="#" class="dropdown-
toggle" data-toggle="dropdown">Welcome, {{Auth::user()->name}} <b
class="caret"></b></a>
                      <ul class="dropdown-menu">
                          <li><a href="/user/orders"><i class="icon-
envelope"></i> My Orders</a></li>
                          <li class="divider"></li>
                          <li><a href="/user/logout"><i class="icon-
off"></i> Logout</a></li>
                      </ul>
                  </li>
                @endif
                </ul>
              </div>
            </div>
        </div>
    </div>
</div>

@yield('content')
```

```
    <script src="http://code.jquery.com/jquery.js"></script>
    <script src="//netdna.bootstrapcdn.com/bootstrap/3.0.0/js/bootstrap.
min.js"></script>
    <script type="text/javascript">
    $(function() {
      $('.dropdown-toggle').dropdown();

      $('.dropdown input, .dropdown label').click(function(e) {
        e.stopPropagation();
      });
    });

    @if(isset($error))
      alert("{{$error}}");
    @endif

    @if(Session::has('error'))
      alert("{{Session::get('error')}}");
    @endif

    @if(Session::has('message'))
      alert("{{Session::get('message')}}");
    @endif

    </script>
  </body>
  </html>
```

The template file contains a menu, a login form, and some JavaScript code for the drop-down login form. We'll use the file as our application's layout template. We need to code our login and logout functions in the `UserController.php` file located at `app/controllers`. The login function should look like the following code:

```
  <?php
  class UserController extends BaseController {

    public function postLogin()
    {
      $email=Input::get('email');
      $password=Input::get('password');

      if (Auth::attempt(array('email' => $email, 'password' =>
  $password)))
      {
```

```
            return Redirect::route('index');

        }else{

          return Redirect::route('index')
            ->with('error','Please check your password & email');
        }
      }

      public function getLogout()
      {
        Auth::logout();
        return Redirect::route('index');
      }
    }
```

As shown in the following code, we need to add routes to our route file, `routes.php`, found under `apps`:

```
Route::get('/', array('as'=>'index','uses'=>'BookController@
getIndex'));
Route::post('/user/login', array('uses'=>'UserController@postLogin'));
Route::get('/user/logout', array('uses'=>'UserController@getLogout'));
```

Creating a template file to list books

Now we need a template file to list books. As mentioned previously, we need to create a template file under `app/views/` and save it as `book_list.blade.php`. This file should look like the following:

```
@extends('main_layout')

@section('content')

<div class="container">
  <div class="span12">
    <div class="row">
      <ul class="thumbnails">
        @foreach($books as $book)
        <li class="span4">
          <div class="thumbnail">
            <img src="/images/{{$book->cover}}" alt="ALT NAME">
            <div class="caption">
              <h3>{{$book->title}}</h3>
```

```
                <p>Author : <b>{{$book->author->name}} {{$book->author-
>surname}}</b></p>
                <p>Price : <b>{{$book->price}}</b></p>
                <form action="/cart/add" name="add_to_cart"
method="post" accept-charset="UTF-8">
                  <input type="hidden" name="book" value="{{$book->id}}"
/>
                  <select name="amount" style="width: 100%;">
                    <option value="1">1</option>
                    <option value="2">2</option>
                    <option value="3">3</option>
                    <option value="4">4</option>
                    <option value="5">5</option>
                  </select>
                <p align="center"><button class="btn btn-info btn-
block">Add to Cart</button></p>
              </form>
              </div>
            </div>
          </li>
          @endforeach
        </ul>
      </div>
    </div>
  </div>

@stop
```

The template file has a form to add books to a cart. Now we need to code our functions in the `CartController.php` file located at `app/controllers/`. The content of `CartController.php` should look like the following:

```
<?php
class CartController extends BaseController {

  public function postAddToCart()
  {
    $rules=array(

      'amount'=>'required|numeric',
      'book'=>'required|numeric|exists:books,id'
    );

    $validator = Validator::make(Input::all(), $rules);
```

```
        if ($validator->fails())
        {
            return Redirect::route('index')->with('error','The book
could not added to your cart!');
        }

        $member_id = Auth::user()->id;
        $book_id = Input::get('book');
        $amount = Input::get('amount');

        $book = Book::find($book_id);
        $total = $amount*$book->price;

         $count = Cart::where('book_id','=',$book_id)->where('member_
id','=',$member_id)->count();

         if($count){

           return Redirect::route('index')->with('error','The book
already in your cart.');
         }

        Cart::create(
          array(
          'member_id'=>$member_id,
          'book_id'=>$book_id,
          'amount'=>$amount,
          'total'=>$total
          ));

        return Redirect::route('cart');
    }


    public function getIndex(){

      $member_id = Auth::user()->id;

      $cart_books=Cart::with('Books')->where('member_id','=',$member_
id)->get();

      $cart_total=Cart::with('Books')->where('member_id','=',$member_
id)->sum('total');
```

```
        if(!$cart_books){

          return Redirect::route('index')->with('error','Your cart is
    empty');
        }

        return View::make('cart')
              ->with('cart_books', $cart_books)
              ->with('cart_total',$cart_total);
      }

      public function getDelete($id){

        $cart = Cart::find($id)->delete();

        return Redirect::route('cart');
      }

    }
```

Our controller has three functions. The first of them is `postAddToCart()`:

```
    public function postAddToCart()
      {
        $rules=array(

          'amount'=>'required|numeric',
          'book'=>'required|numeric|exists:books,id'
        );

        $validator = Validator::make(Input::all(), $rules);

          if ($validator->fails())
          {
              return Redirect::route('index')->with('error','The book
    could not added to your cart!');
          }

          $member_id = Auth::user()->id;
          $book_id = Input::get('book');
          $amount = Input::get('amount');

          $book = Book::find($book_id);
          $total = $amount*$book->price;
```

```
        $count = Cart::where('book_id','=',$book_id)->where('member_
id','=',$member_id)->count();

        if($count){

          return Redirect::route('index')->with('error','The book
already in your cart.');
        }

       Cart::create(
         array(
         'member_id'=>$member_id,
         'book_id'=>$book_id,
         'amount'=>$amount,
         'total'=>$total
         ));

       return Redirect::route('cart');
   }
```

The function basically, at first, validates the posted data. The validated data checks the `carts` table for duplicate records. If the same book is not in the member's cart, the function creates a new record in the `carts` table. The second function of the `CartController` is `getIndex()`:

```
   public function getIndex(){

     $member_id = Auth::user()->id;

     $cart_books=Cart::with('Books')->where('member_id','=',$member_
id)->get();

     $cart_total=Cart::with('Books')->where('member_id','=',$member_
id)->sum('total');

     if(!$cart_books){

       return Redirect::route('index')->with('error','Your cart is
empty');
     }

     return View::make('cart')
           ->with('cart_books', $cart_books)
           ->with('cart_total',$cart_total);
   }
```

The function fetches whole cart items, books' information, and cart total and passes the data to the template file. The last function of the class is `getDelete()`:

```
public function getDelete($id){

  $cart = Cart::find($id)->delete();

  return Redirect::route('cart');
}
```

The function basically finds from the `carts` table the given ID and deletes the record. We use the function to delete items from a cart. Now we need to create a template file. The file will show all cart information of members and also contains the order form. Save the file under `app/views/` as `cart.blade.php`. The content of `cart.blade.php` should look like the following:

```
@extends('main_layout')

@section('content')

<div class="container" style="width:60%">
  <h1>Your Cart</h1>
  <table class="table">
    <tbody>
      <tr>
        <td>
          <b>Title</b>
        </td>
        <td>
          <b>Amount</b>
        </td>
        <td>
          <b>Price</b>
        </td>
        <td>
          <b>Total</b>
        </td>
        <td>
          <b>Delete</b>
        </td>
      </tr>
      @foreach($cart_books as $cart_item)
        <tr>
          <td>{{$cart_item->Books->title}}</td>
          <td>
```

```
            {{$cart_item->amount}}
          </td>
          <td>
            {{$cart_item->Books->price}}
          </td>
          <td>
            {{$cart_item->total}}
          </td>
          <td>
            <a href="{{URL::route('delete_book_from_cart',array($cart_
item->id))}}">Delete</a>
          </td>
        </tr>
      @endforeach
      <tr>
        <td>
        </td>
        <td>
        </td>
        <td>
          <b>Total</b>
        </td>
        <td>
          <b>{{$cart_total}}</b>
        </td>
        <td>
        </td>
      </tr>
    </tbody>
  </table>
  <h1>Shipping</h1>
  <form action="/order" method="post" accept-charset="UTF-8">
    <label>Address</label>
    <textarea class="span4" name="address" rows="5"></textarea>
    <button class="btn btn-block btn-primary btn-large">Place order</
button>
  </form>
</div>
@stop
```

Now we need to write our routes. The functions of the controller should just be accessible to members. So we can easily use Laravel's built-in `auth.basic` filter:

```
Route::get('/cart', array('before'=>'auth.basic','as'=>'cart','uses'=>
'CartController@getIndex'));
```

```
Route::post('/cart/add', array('before'=>'auth.
basic','uses'=>'CartController@postAddToCart'));
Route::get('/cart/delete/{id}', array('before'=>'auth.
basic','as'=>'delete_book_from_cart','uses'=>'CartController@
getDelete'));
```

Taking orders

As you may remember, we've already created an order form in the `cart.blade.php` template file located at `app/views/`. Now we need to process the order. Let's code the `OrderController.php` file under `app/controllers/`:

```
<?php
class OrderController extends BaseController {

  public function postOrder()
  {
    $rules=array(

      'address'=>'required'
    );

  $validator = Validator::make(Input::all(), $rules);

      if ($validator->fails())
      {
          return Redirect::route('cart')->with('error','Address field
is required!');
      }

      $member_id = Auth::user()->id;
      $address = Input::get('address');

       $cart_books = Cart::with('Books')->where('member_
id','=',$member_id)->get();

       $cart_total=Cart::with('Books')->where('member_id','=',$member_
id)->sum('total');

       if(!$cart_books){

         return Redirect::route('index')->with('error','Your cart is
empty.');
       }
```

```
        $order = Order::create(
          array(
          'member_id'=>$member_id,
          'address'=>$address,
          'total'=>$cart_total
          ));

        foreach ($cart_books as $order_books) {

          $order->orderItems()->attach($order_books->book_id, array(
            'amount'=>$order_books->amount,
            'price'=>$order_books->Books->price,
            'total'=>$order_books->Books->price*$order_books->amount
            ));

        }

        Cart::where('member_id','=',$member_id)->delete();

        return Redirect::route('index')->with('message','Your order
processed successfully.');
    }


    public function getIndex(){

      $member_id = Auth::user()->id;

      if(Auth::user()->admin){

        $orders=Order::all();

      }else{

        $orders=Order::with('orderItems')->where('member_
id','=',$member_id)->get();
      }

      if(!$orders){

        return Redirect::route('index')->with('error','There is no
order.');
      }
```

```
        return View::make('order')
              ->with('orders', $orders);
    }
}
```

The controller has two functions. The first of them is `postOrder()`:

```
public function postOrder()
  {
    $rules=array(

      'address'=>'required'
    );

  $validator = Validator::make(Input::all(), $rules);

      if ($validator->fails())
      {
          return Redirect::route('cart')->with('error','Address field
is required!');
      }

      $member_id = Auth::user()->id;
      $address = Input::get('address');

       $cart_books = Cart::with('Books')->where('member_
id','=',$member_id)->get();

       $cart_total=Cart::with('Books')->where('member_id','=',$member_
id)->sum('total');

       if(!$cart_books){

         return Redirect::route('index')->with('error','Your cart is
empty.');
       }

      $order = Order::create(
        array(
        'member_id'=>$member_id,
        'address'=>$address,
        'total'=>$cart_total
        ));

      foreach ($cart_books as $order_books) {
```

```
        $order->orderItems()->attach($order_books->book_id, array(
          'amount'=>$order_books->amount,
          'price'=>$order_books->Books->price,
          'total'=>$order_books->Books->price*$order_books->amount
          ));

      }

      Cart::where('member_id','=',$member_id)->delete();

      return Redirect::route('index')->with('message','Your order
processed successfully.');
  }
```

The function, first, validates the posted data. After successful validation, the function creates a new order on the `orders` table. The `order` table stores the member ID, shipping address, and total amount of the order. Then, the function attaches all cart items to the pivot table with their amount, price, and total amounts. In this way, the order items will not be affected by any price change. The function then deletes all items from the member's cart. The second function of the controller is `getIndex()`:

```
public function getIndex(){

    $member_id = Auth::user()->id;

    if(Auth::user()->admin){

      $orders=Order::all();

    }else{

      $orders=Order::with('orderItems')->where('member_
id','=',$member_id)->get();
    }

    if(!$orders){

      return Redirect::route('index')->with('error','There is no
order.');
    }

    return View::make('order')
          ->with('orders', $orders);
  }
```

The function queries the database by looking at current user rights. If the current user has admin rights, the function fetches all the orders. If the current user has no admin rights, the function fetches just the user's orders. So, now we need to write our routes. Add the following route code to `app/routes.php`:

```
Route::post('/order', array('before'=>'auth.
basic','uses'=>'OrderController@postOrder'));
Route::get('/user/orders', array('before'=>'auth.
basic','uses'=>'OrderController@getIndex'));
```

Our e-commerce application is almost done. Now we need to add a template file. Save the file under `app/views/` as `cart.blade.php`. The content of `cart.blade.php` should be like the following:

```
@extends('main_layout')
@section('content')
<div class="container" style="width:60%">
<h3>Your Orders</h3>
<div class="menu">
  <div class="accordion">
@foreach($orders as $order)
 <div class="accordion-group">
      <div class="accordion-heading country">
        @if(Auth::user()->admin)
        <a class="accordion-toggle" data-toggle="collapse"
href="#order{{$order->id}}">Order #{{$order->id}} - {{$order->User-
>name}} - {{$order->created_at}}</a>
        @else
        <a class="accordion-toggle" data-toggle="collapse"
href="#order{{$order->id}}">Order #{{$order->id}} - {{$order->created_
at}}</a>
        @endif
      </div>
      <div id="order{{$order->id}}" class="accordion-body collapse">
        <div class="accordion-inner">
          <table class="table table-striped table-condensed">
            <thead>
              <tr>
              <th>
              Title
              </th>
              <th>
              Amount
              </th>
              <th>
```

```
                Price
                </th>
                <th>
                Total
                </th>
                </tr>
              </thead>
              <tbody>
              @foreach($order->orderItems as $orderitem)
                <tr>
                  <td>{{$orderitem->title}}</td>
                  <td>{{$orderitem->pivot->amount}}</td>
                  <td>{{$orderitem->pivot->price}}</td>
                  <td>{{$orderitem->pivot->total}}</td>
                </tr>
              @endforeach
                <tr>
                  <td></td>
                  <td></td>
                  <td><b>Total</b></td>
                  <td><b>{{$order->total}}</b></td>
                </tr>
                <tr>
                  <td><b>Shipping Address</b></td>
                  <td>{{$order->address}}</td>
                  <td></td>
                  <td></td>
                </tr>
              </tbody>
            </table>
          </div>
        </div>
      </div>
  @endforeach
  </div>
  </div>
  @stop
```

The template file contains all the information about orders. The template is a very simple example of how to use pivot table columns. The pivot data comes as an array. So, we've used the `foreach` loop to use the data. You can store any data that you do not want to be affected by any changes in the database, such as price changes.

Summary

In this chapter, we've built a simple e-commerce application. As you can see, thanks to Laravel's template system and built-in authorization system, you can easily create huge applications. You can improve the application with third-party packages. Since Laravel Version 4, the main package manager has been Composer. There is a huge library at `http://packagist.org`, which provides packages for image manipulating, social media APIs, and so on. The number of packages increase day-by-day, which are becoming, by default, Laravel compatible. We suggest that, before coding anything, you take a look at the Packagist website. There are many contributors still sharing their codes while you're reading these sentences. Reviewing another programmer's code gives new answers to old problems. Do not forget to share your knowledge with people to have a better programming experience.

During the entire book we've tried to explain building different kinds of applications with the Laravel PHP framework. So, we've covered RESTful controllers, routing, route filters, the authentication class, Blade template engine, database migrations, database seeding, string, and file-processing classes. Also, we've given some tips for rapid development with Laravel. We hope that this book will be a good resource for learning the Laravel framework.

The co-author of the book, *Halil İbrahim Yılmaz*, has developed an open source, multilingual CMS with Laravel, which is named HERKOBI. You can use both the source codes of the book chapters and the source code of the CMS. You can access the CRM and the codes at `http://herkobi.org` and also at `http://herkobi.com`.

Laravel has a good community, which is very helpful and friendly. You can ask any question in the Laravel forums. The international Laravel community can be accessed at `http://laravel.com`. Laravel also has some national Laravel communities, such as a Turkish Laravel community, which is located at `http://laravel.gen.tr`.

You can send an e-mail to the authors when you need help on anything in the book or the Laravel PHP framework.

Thank you for being interested in and purchasing this book.

Index

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Thank you for buying
Laravel Application Development Blueprints

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

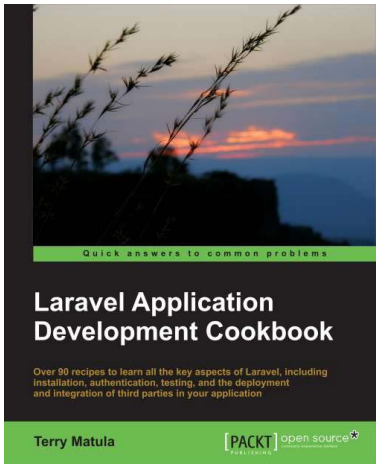
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] open source
community experience distilled
PUBLISHING



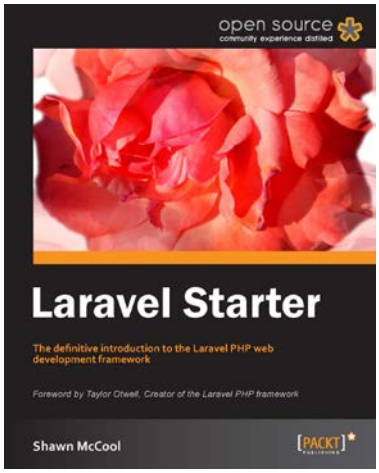
Laravel Application Development Cookbook

ISBN: 978-1-78216-282-7 Paperback: 272 pages

Over 90 recipes to learn all the key aspects of Laravel, including installation, authentication, testing, and the deployment and integration of third parties in your application

1. Install and set up a Laravel application and then deploy and integrate third parties in your application
2. Create a secure authentication system and build a RESTful API
3. Build your own Composer Package and incorporate JavaScript and Ajax methods into Laravel

---



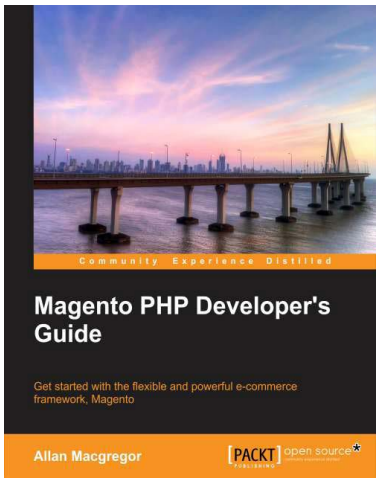
Instant Laravel Starter

ISBN: 978-1-78216-090-8 Paperback: 64 pages

The definitive introduction to the Laravel PHP web-development framework

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results.
2. Create databases using Laravel's migrations
3. Learn how to implement powerful relationships with Laravel's own "Eloquent" ActiveRecord implementation
4. Learn about maximizing code reuse with the bundles
5. Get started by building a useful real-world application

[PACKT] open source
community experience distilled
PUBLISHING

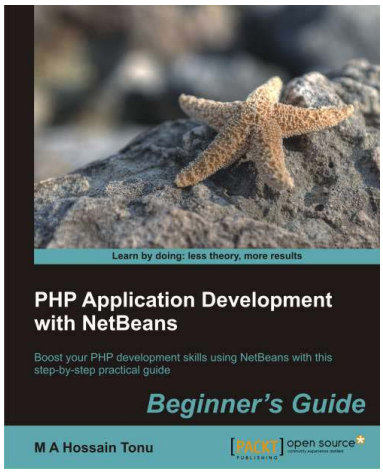


Magento PHP Developer's Guide

ISBN: 978-1-78216-306-0 Paperback: 256 pages

Get started with the flexible and powerful e-commerce framework, Magento

1. Build your first Magento extension, step-by-step
2. Extend core Magento functionality, such as the API
3. Learn how to test your Magento code



PHP Application Development with NetBeans: Beginner's Guide

ISBN: 978-1-84951-580-1 Paperback: 302 pages

Boost your PHP development skills with this step-by-step practical guide

1. Clear, step-by-step instructions with lots of practical examples
2. Develop cutting-edge PHP applications like never before with the help of this popular IDE, through quick and simple techniques
3. Experience exciting features of PHP application development with real-life PHP projects