The PHPLeaks Developer’s Guide

W. Jason Gilmore

The PHPLeaks Developer’s Guide

W. Jason Gilmore

Contents

Building PHPLeaks: a Concise Guide

PHPLeaks (http://www.phpleaks.com[1]) is a link aggregation site for the PHP community, created to help fellow developers quickly find the latest and greatest tutorials, development resources, and other useful news. In addition to providing registered users with a simple form for sharing links with fellow developers, users may also favorite links for later review. The site also tracks the number of times users follow outbound links so as to determine link popularity.

In this guide I'll offer a brief tour through many of the key steps taken to develop the project. Although I'll be regularly updating and expanding this book, keep in mind not every feature is discussed in detail so if you have any questions feel free to e-mail me at wj@wjgilmore.com. Also, although PHPLeaks is functional it remains very much a work in progress, so stay tuned for an ongoing stream of changes as I continue adding new features.

The PHPLeaks.com project and this companion developer's guide are *beta releases*. Neither is perfect nor feature complete, which is why it is currently being sold at a discount from the final price. If you find a bug then by all means please e-mail me at wj@wjgilmore.com.

About this Guide

This guide is broken into five chapters, each of which is briefly described below.

Chapter 1. Creating and Configuring the PHPLeaks Application

In this opening chapter we'll create the project, configure the database and mail delivery, and install and configure a few key packages used for testing and debugging.

Chapter 2. Integrating Bootstrap, Less CSS, and CoffeeScript

Laravel 5 is bundled with a powerful Gulp[2] API known as Elixir which among other things simplifies the integration of the Less[3] CSS pre-processor and CoffeeScript[4]. In this chapter I'll show you how to configure Elixir to integrate the Bootstrap[5] framework via a Less style sheet and additionally compile CoffeeScript into JavaScript.

[1]http://www.phpleaks.com
[2]http://gulpjs.com/
[3]http://lesscss.org/
[4]http://coffeescript.org/
[5]getbootstrap.com/

Chapter 3. Creating the Category and Link Functionality

PHPLeaks revolves around the presentation of web links supplied by the PHP community. In this chapter I'll talk about how the `Link` and `Category` models are associated, and discuss in great detail how the link submission form was implemented. You'll also learn how to create sluggable URLs which use friendly strings as parameters instead of integers.

Chapter 4. Integrating User Accounts

PHPLeaks wouldn't really have any reason to exist were it not for users' ability to create an account and add, find and track links. Fortunately it's incredibly easy to integrate user registration, sign in and account profile features into a Laravel 5 application, and in this chapter I'll touch upon the various changes made to these native features to suit PHPLeaks.

Chapter 5. Creating an Administration Console

In this final chapter I'll show you how to create a restricted administration console which you can use to review user accounts, easily edit submitted links, ban users and perform other administrative tasks.

About the Author

W. Jason Gilmore[6] is a software developer, consultant, and bestselling author. He has spent much of the past 15 years helping companies of all sizes build amazing solutions. Recent projects include a SaaS for the architecture and interior design industries, an e-commerce analytics application for a globally recognized publisher, a Linux-powered autonomous environmental monitoring buoy, and a 10,000+ product online store.

Jason is the author of seven books, including the bestselling "Beginning PHP and MySQL, Fourth Edition", "Easy Active Record for Rails Developers", and "Easy PHP Websites with the Zend Framework, Second Edition".

Over the years Jason has published more than 300 articles within popular publications such as Developer.com, JSMag, and Linux Magazine, and instructed hundreds of students in the United States and Europe. Jason is co-founder of the wildly popular CodeMash Conference[7], the largest multi-day developer event in the Midwest.

Away from the keyboard, you'll often find Jason playing with his kids, hunched over a chess board, and having fun with DIY electronics.

Jason loves talking to readers and invites you to e-mail him at wj@wjgilmore.com.

[6] http://www.wjgilmore.com
[7] http://www.codemash.org

Errata and Suggestions

Nobody is perfect, particularly when it comes to writing about technology. I've surely made some mistakes in both code and grammar, and probably completely botched more than a few examples and explanations. If you would like to report an error, ask a question or offer a suggestion, please e-mail me at wj@wjgilmore.com.

Chapter 1. Creating and Configuring the PHPLeaks Application

PHPLeaks takes advantage of several different third-party packages for implementing features such as friendly URLS, debugging and testing. In this chapter I'll highlight the steps taken to create the project and configure these various packages. Let's kick things off by building the application.

Building and Configuring the Application

Begin by creating the new Laravel application. In this example I'm using the Laravel Installer however other project creation solutions exist which you can learn about here[8]:

```
$ laravel new dev.phpleaks.com
Crafting application...
Application ready! Build something amazing.
```

After creating the application you can set the namespace:

```
$ php artisan app:name Phpleaks
Application namespace set!
```

Next, create a copy of the `.env.example` file, renaming the copy to `.env`:

```
$ cp .env.example .env
```

Your application will recognize `.env` as the configuration file, meaning you'll want to use it to manage any configuration settings pertinent to your development environment. Note this file is *not* managed in version control (open the `.gitignore` file and you'll see that `.env` is listed there) so as to ensure sensitive configuration data is not potentially distributed via the repository. We'll update this file next to include both database- and mail delivery-related configuration settings.

Configuring the Database

PHPLeaks is logically backed by a database and so you'll want to create a database for use in your development environment. Laravel supports MySQL, PostgreSQL, SQLite, and Microsoft SQL Server, so any of these solutions will do. I happen to use MySQL, and so will create the database by logging into the MySQL client and executing the following command:

[8]http://laravel.com/docs/5.0/installation

```
mysql> create database dev_phpleaks;
```

After creating the database you'll want to create a user which your application will use to interact with the database. If you're using MySQL this is most easily done using the `GRANT` command so have a look at the MySQL documentation for more details. After creating the database and user open up your `.env` file and update the `DB_HOST`, `DB_DATABASE`, `DB_USERNAME` and `DB_PASSWORD` settings to reflect the appropriate values. If you're using a database other than MySQL you'll additionally need to open the `config/database.php` file and change the `default` setting from the default `mysql` value to either `sqlite` (SQLite), `pgsql` (PostgreSQL), or `sqlsrv` (Microsoft SQL Server).

After updating and saving the `.env` file, you'll want to run the initial set of migrations included with every new Laravel application. These migrations create database tables used for managing registered users and password resets. To run the migrations execute the following command from your project's root directory:

```
$ php artisan migrate
Migration table created successfully.
Migrated: 2014_10_12_000000_create_users_table
Migrated: 2014_10_12_100000_create_password_resets_table
```

Configuring Mail Delivery Using Mandrill

Laravel includes support for a number of different e-mail delivery solutions, including Sendmail, PHP's `mail()` function, and third-party providers such as Mailgun and Mandrill. I very much prefer to not deal with the hassle of managing mail-related matters myself and so leave the matter to the experts at Mandrill[9]. Mandrill is an e-mail delivery service created by the same company behind MailChimp[10]. It's free for sending up to 10,000 e-mails per month, which is plenty.

If you want to use Mandrill, you'll first need to sign up for an account over at https://mandrill.com/[11]. Once registered you'll be provided with an API key. Open your `.env` file and add that API key along with other pertinent information to the file:

[9]https://mandrill.com/
[10]http://mailchimp.com
[11]https://mandrill.com/

```
MAIL_DRIVER=mandrill
MAIL_HOST=smtp.mandrillapp.com
MAIL_PORT=587
MAIL_FROM=YOUR_MANDRILL_EMAIL_HERE
MAIL_NAME=EMAIL_FROM_NAME
MAIL_USERNAME=YOUR_MANDRILL_USERNAME_HERE
MAIL_PASSWORD=SECRET_API_KEY_HERE
```

Next, open up the PHPLeaks `config/mail.php` file and see how I updated the `from` array to use the `.env` file's `MAIL_FROM` and `MAIL_NAME` settings. Save those changes and update the `config/services.php` file to identify the API KEY:

```
'mandrill' => [
        'secret' => env('MAIL_PASSWORD'),
],
```

With these changes in place you can begin sending e-mail through your Laravel application using Mandrill. You can easily test the service integration out by creating an account (navigate to `/auth/register`), signing out, and then initiating the password recovery process (navigate to `/auth/password`). If everything is properly configured an e-mail will be sent to the address associated with your newly created account. If the e-mail does not arrive or you receive an error, the problem almost certainly has to do with your configuration, so return to the `.env`, `config/mail.php` and `config/services.php` files and update any errant settings.

Installing and Configuring Third-Party Packages

With the application created let's install and configure a few third-party packages. None of these packages are strictly required to create a successful Laravel application but I personally find them to be both convenient and a big time saver.

Install laravelcollective/html

Laravel's HTML and form builders were removed from the framework as of version 5.0 and made available via a separate package. While I don't typically use the HTML builders I'm a big fan of the form builders if for any other reason it formalizes the syntax used to generate the form HTML. To install the HTML package (which includes both the HTML and form builders) execute the following command:

```
$ composer require "laravelcollective/html:~5.0"
```

Once installed, open `config/app.php` and add the following line to the `providers` array:

```
'Collective\Html\HtmlServiceProvider',
```

Scroll further down the `config/app.php` file and add the following line to the `aliases` array:

```
'HTML' => 'Collective\Html\HtmlFacade'
```

Save the changes and you're ready to begin using the HTML and form builders! We'll return to these builders later in the guide.

Installing the Laravel Debugbar

The Laravel Debugbar[12] can save you a bunch of time and hassle when debugging your application. To install it execute the following command from inside your project's root directory:

```
$ composer require "barryvdh/laravel-debugbar:~2.0"
```

Next, add the following lines to your `config/app.php` `$providers` and `$aliases` arrays, respectively:

```
'providers' => [
        ...
    'Barryvdh\Debugbar\ServiceProvider',
]

'aliases' => [
        ...
        'Debugbar' => 'Barryvdh\Debugbar\Facade'
]
```

Save these changes and when you load the application into your browser you should see the Debugbar console at the bottom of the screen! This is an incredibly useful tool for monitoring queries, requests, routing, and other useful metrics.

Installing and Configuring Codeception

Laravel 5 supports PHPUnit[13] and phpspec[14] by default. I haven't had the opportunity to use phpspec and while I like using PHPUnit for testing models I've lately prefer to use Codeception[15] for functional and acceptance testing. If you'd like to begin using Codeception to test your Laravel applications follow these instructions. Begin by adding the Codeception package to your project:

[12] https://github.com/barryvdh/laravel-debugbar
[13] https://phpunit.de/
[14] http://www.phpspec.net/
[15] http://codeception.com/

```
$ composer require --dev "codeception/codeception:*"
```

Next, install the Codeception Laravel 5 module. This module is required so Codeception knows how to properly interact with your application. You can learn more about the module here[16]:

```
$ composer require --dev janhenkgerritsen/codeception-laravel5
```

With Codeception and the Laravel 5 module installed, you'll want to create the various directories and configuration files used by Codeception to manage your project tests:

```
$ vendor/bin/codecept bootstrap
```

Once this command completes execution, you'll find a number of new files and directories inside your Laravel application's `tests` directory. We'll next need to update a few of these files.

Update the Configuration Files

I am currently having issues properly configuring Codeception to create and execute Laravel-driven acceptance tests (suggestions welcome) and so at this point in time have only added functional tests to the application. To configure Codeception to support functional testing of your Laravel application, begin by creating a test database. I'm already using MySQL for development purposes and so would prefer to use MySQL for the test database as well. To do so I've created a database named `test_phpleaks`. Next, open up the file `codeception.yml` (it resides in your project's root directory) and update the `dsn`, `user`, and `password` settings accordingly:

```
modules:
    config:
        Db:
            dsn: 'mysql:host=localhost;dbname=test_phpleaks'
            user: 'phpleaks_user'
            password: 'secret'
            dump: tests/_data/dump.sql
            populate: true
            cleanup: true
```

I wish there were a way to pull that database configuration information in via an environment configuration file, and maybe indeed there is (let me know if so). However I'm not too concerned about exposing this information even if it is included in the source repository since the database is used solely for testing purposes and does not contain sensitive data of any sort.

[16]https://github.com/janhenkgerritsen/codeception-laravel5

The `dump` setting identifies the location of a raw SQL database export containing any data you'd like to import into the test database following any migrations. I'm really not a fan of this approach because it strikes me as overly awkward. Alternatively I suggest having a look at FactoryMuffin[17]. The `populate` setting tells Codeception to load the data export before the tests begin running. The `cleanup` setting indicates whether the data export should be reloaded after each test.

Next, open up `tests/functional.suite.yml` and update it to look like this:

```
class_name: FunctionalTester
modules:
    enabled: [Db, Filesystem, FunctionalHelper, Laravel5]
    config:
        Laravel5:
            environment_file: .env.testing
```

Specifically you're enabling the `Db` and `Laravel5` modules, and then specifying that the Laravel 5 module should use the environment file `.env.testing`. You'll need to create this file, copying your `.env` file to `.env.testing` and then updating the settings accordingly. You should also add this file to `.gitignore` although for demonstration purposes I've opted to leave it in the repository.

With these configuration changes in place you should be able to create and execute a functional test skeleton:

```
$ vendor/bin/codecept generate:cest functional Welcome
$ vendor/bin/codecept run

Codeception PHP Testing Framework v2.0.11
Powered by PHPUnit 4.5.0 by Sebastian Bergmann and
contributors.

Acceptance Tests (0)
-----------------------------
-----------------------------

Functional Tests (1)
-----------------------------
Try to test (WelcomeCest::tryToTest)                                  \
          Ok
-----------------------------

Unit Tests (0)
-----------------------------
```

[17] http://codeception.com/10-23-2014/managing-data-with-factorymuffin.html

```
-----------------------------

Time: 291 ms, Memory: 15.75Mb

OK (1 test, 0 assertions)
```

Be sure to have a look at the tests found inside `tests/functional`. Among other things I show you how to create a reusable registration method (inside `tests/_support/FunctionalHelper.php`), confirm various parts of the application are accessible and contain specific content, and confirm proper functioning of form helpers. This is still very much a work in progress and I hope to add at least a dozen other tests in the near future.

Summary

I'll be the first to admit that project configuration isn't exactly scintillating discussion however completing these mundane tasks allow you to have much more fun during the actual development process because testing and debugging isn't such a drag. With these configurations and packages in place we're ready to move on to far more exciting features!

Chapter 2. Integrating Bootstrap, Less CSS, and CoffeeScript

I'm a fan of sorting out a web application's layout and CSS early on in the development process so I can continually improve and adjust the design throughout the entire course of the project. Given my horrible design skills, it shouldn't come as a surprise that I require as much time as possible to make improvements of this nature.

In this chapter I'll highlight some of the steps taken to integrate a Bootstrap[18]-driven layout into PHPLeaks. Additionally, I'll show you how Elixir was configured to easily compile the Less[19] style sheet and CoffeeScript.

Creating the Application Layout

Laravel 5 includes a default layout found in `resources/views/app.blade.php`. This layout is used in conjunction with the user account-related views bundled into every new Laravel 5 application by default (the views found in `resources/views/auth`). While there's nothing wrong with this default layout I prefer to build layouts from scratch and so you'll find PHPLeaks' layout inside `resources/views/layouts` within a file named `master.blade.php` (in keeping with the traditional Laravel 4 naming convention). If you have a look at the various views found in `resources/views` you'll see they all extend from this `master.blade.php` layout:

```
@extends('layouts.master')

@section('content')

...

@endsection
```

Inside the layout header you'll see references to a stylesheet named `app.css` and a JavaScript file named `application.js`:

[18] http://getbootstrap.com/
[19] http://lesscss.org

```
<link rel="stylesheet" href="/css/app.css">
...
<script src="/js/application.js"></script>
```

Neither of these files exist by default. While you certainly could create these files from scratch there is a much better way in which you'll treat the project CSS and JavaScript as *assets* which are managed using Less[20] and CoffeeScript[21]. If you're not familiar with these technologies I suggest spending some time learning more about them as both will likely save you a fair bit of time and sanity in future projects. In the next section I'll show you how to use Laravel Elixir to integrate a convenient Less and CoffeeScript compilation workflow into your project.

Installing and Configuring Elixir

Laravel Elixir[22] can eliminate many mundane repetitive tasks associated with modern web application development, such as Less CSS and CoffeeScript compilation. Logically you would always want to have the most recent compiled versions of the Less and CoffeeScript source files available to your application, yet to do so without automation means you would have to constantly return to the terminal to run the respective compilers (typically `lessc` and `coffee`).

As of Laravel 5 you can automate the process of compiling these asset files every time they change by taking advantage of the new Elixir API. Elixir is a Gulp[23] extension, offering Laravel-specific syntax that allows you to easily create tasks for compiling Less and CoffeeScript, running PHPUnit tests, and performing other tedious, repetitive tasks.

As I mentioned, Elixir is based on Gulp, and Gulp is based on Node.js[24], meaning you'll need to install Node.js on your development machine. This is easily done by downloading one of the installers via the Node.js website[25]. If you'd prefer to build Node from source you can download the source code via this link. If like me you're a Mac user, you can install Node via Homebrew. Linux users additionally likely have access to Node via their distribution's package manager.

Once installed you can confirm Node is accessible via the command-line by displaying the Node version number:

```
$ node -v
v0.10.36
```

With Node installed it's time to install Elixir! Node users have access to a great number of third-party libraries known as Node Packaged Modules (NPM). You can install these modules (including Gulp and Elixir) via the aptly-named `npm` utility. To do so you'll enter the project root directory and execute the following command:

[20]http://lesscss.org
[21]http://coffeescript.org
[22]http://laravel.com/docs/5.0/elixir
[23]http://gulpjs.com/
[24]https://nodejs.org/
[25]http://nodejs.org/download/

```
$ npm install
```

When `npm install` is executed, `npm` will look for a file named `package.json` (just as Composer looks for `composer.json`) and subsequently install any dependencies. If you open `package.json` you'll see two dependencies are specified:

```
{
  "private": true,
  "devDependencies": {
    "gulp": "^3.8.8",
    "laravel-elixir": "*"
  }
}
```

As you can see, both `gulp` and `laravel-elixir` are specified, meaning once the npm installer is finished you'll be able to take advantage of these packages. I'll show you how to do that next.

Compiling Less with Elixir

Fortunately doing so is very easy, because all you have to do to run the default set of Elixir tasks found in `gulpfile.js` is run the following command:

```
$ gulp
[11:18:23] Using gulpfile ~/Software/trash.phpleaks.com/gulpfile.js
[11:18:23] Starting 'default'...
[11:18:23] Starting 'less'...
[11:18:24] Finished 'default' after 514 ms
[11:18:24] gulp-notify: [Laravel Elixir]
[11:18:24] Finished 'less' after 634 ms
```

So what happened here? The lone Elixir task found in `gulpfile.js` was executed:

```
elixir(function(mix) {
    mix.less('app.less');
});
```

This task tells Elixir to compile the file named `app.less`. This `app.less` Less file is included by default in every new Laravel application, and it's found in `resources/assets/less`. One great aspect of Elixir is you don't have to remember the location of these assets; the default locations are already known to Elixir. The default `app.less` file is empty, meaning the compiled CSS file (found in `public/css/app.css`) is also empty, but we'll remedy that soon enough. Even so, let's make this empty CSS file available to the layout now by adding the following line to the layout header:

```
<link rel="stylesheet" href="/css/app.css">
```

Next I'll show you how I integrated Bootstrap into the Less workflow, but first I want to share a very important trick in which you can use the Gulp command `watch` to automatically run the tasks every time one of the target files changes. Open a new terminal tab and execute the following command:

```
$ gulp watch
[12:00:19] Using gulpfile ~/Software/trash.phpleaks.com/gulpfile.js
[12:00:19] Starting 'watch'...
[12:00:19] Starting 'less'...
[12:00:20] Finished 'watch' after 522 ms
[12:00:20] gulp-notify: [Laravel Elixir]
[12:00:20] Finished 'less' after 629 ms
[12:00:20] Starting 'watch-assets'...
[12:00:20] Finished 'watch-assets' after 7.83 ms
```

Leave this command running and keep your eye on the compiled CSS file `public/css/app.css` as we integrate Bootstrap in the next section.

Integrating Bootstrap

The Bootstrap[26] source files were initially included with each new Laravel application but they were more recently removed, meaning you'll need to download the source files yourself in order to incorporate them into the Less workflow. This is really easy to do though, just head over to http://getbootstrap.com/getting-started/[27] and download the *source code*. Do not download the compiled CSS otherwise you won't be able to easily override Bootstrap's stylistic defaults using Less. Unzip the zip file into a directory named `bootstrap`, and place this directory inside `resources/assets/less`. Then open `app.less` and add the following line to the top of the file:

```
@import "bootstrap/less/bootstrap";
```

If you're still running `gulp watch` then the compilation process will begin immediately. After the compilation is complete, have another look at the `public/css/app.css` file and you'll see that all of the Bootstrap source files found in `bootstrap/less` were compiled and concatenated together into the `public/css/app.css` file. With this in place you can return to `app.less` and begin selectively overriding Bootstrap's default stylistic settings. For instance the following modified `app.less` file will set Bootstrap's navigation bar background color to `#374140` and change the bar height to 60 pixels:

[26]http://getbootstrap.com/
[27]http://getbootstrap.com/getting-started/

```
@import "bootstrap/bootstrap";

@navbar-default-bg: #374140;

@navbar-height: 60px;
```

Have a look at the PHPLeak's `app.less` and `master.blade.php` files for examples of Bootstrap integration and stylistic overrides using Less syntax.

Integrating CoffeeScript

I've never had much fun programming in JavaScript until CoffeeScript[28] came along. Created by JavaScript wizard Jeremy Ashkenas[29], CoffeeScript is simply put, a language that compiles into JavaScript. It looks like JavaScript, but without all of the annoyances that make JavaScript so painful to write. If you're not familiar with CoffeeScript syntax, I think the following simple example found on the CoffeeScript website sums up the syntactical improvement over JavaScript quite nicely. Let's start with the JavaScript:

```
if (typeof elvis !== "undefined" && elvis !== null) {
  alert("I knew it!");
}
```

Wow that is a mess. Here's the same syntax in CoffeeScript:

```
alert "I knew it!" if elvis?
```

Because CoffeeScript is compiled into JavaScript, the above compact statement will automatically be turned into the mess required by the browser's JavaScript engine.

PHPLeaks uses a few dozen lines of CoffeeScript to perform tasks such as providing real-time updates of the number of characters remaining when inserting a link description, and updating the number of times a link has been followed before a clicking user exits the PHPLeaks.com website. You'll find this CoffeeScript file in `resources/assets/coffee/application.coffee`. With a CoffeeScript file in place, you can instruct Elixir to compile it for you, placing the compiled contents in `public/js`. Elixir supports a few different syntactical approaches to doing so, however the most succinct simply involves chaining `coffee()` to the `less()` method as demonstrated here:

[28]http://coffeescript.org/
[29]https://twitter.com/jashkenas

```
elixir(function(mix) {
    mix.less('app.less').coffee();
});
```

Once compiled you'll need to reference the JavaScript file in the layout header:

```
<script src="/js/application.js"></script>
```

Summary

This short but sweet overview hardly scratches the surface in terms of what Elixir is capable of, however I think this makes the case for incorporating it into your workflow even stronger. Consider that with just a bit of configuration and a few lines of code I was able to integrate Bootstrap with Less-driven stylistic overrides and take advantage of CoffeeScript to introduce highly dynamic behaviors into the PHPLeaks website!

Chapter 3. Creating the Category and Link Functionality

PHPLeaks is based entirely around the simple premise of link aggregation and findability. A pretty simple implementation is currently in place, but it's important to lay a proper foundation so as to ensure these capabilities can be easily and effectively expanded in the future. In this chapter I'll highlight a few of the key steps taken to implement the various link-related features and hopefully shed some light into the thought process guiding the implementation decisions.

Integrating Categories

Each link is associated with a single category. While I certainly could have chosen a slightly more complex solution involving associating a link with multiple categories, I decided it would at least initially be more desirable to constrain users to select a single category that most accurately suits the link content, if for any other reason to discourage category spamming. In this section I'll walk through the steps I took to integrate categories into the site.

Creating the Category Model

I kicked things off by creating the `Category` model and associated migration. This is a very simple model and underlying table, consisting of just an ID and name:

```
$ php artisan make:model Category
Model created successfully.
Created Migration: 2015_03_08_031234_create_categories_table
```

Next, open up the newly created migration found in `database/migrations/` and modify the `up()` method to look like this:

```
Schema::create('categories', function(Blueprint $table)
{
    $table->increments('id');
    $table->string('name');
    $table->timestamps();
});
```

With the `name` column added to the migration, run the migration to create the table:

```
$ php artisan migrate
```

With the `Category` model and corresponding table in place let's seed the table with a few initial categories.

Seeding the Category Table

Although it's easy enough to create a restricted form which allows an administrator to add new categories (see Chapter 6 for more information about how I created PHPLeaks' administration console), it made more sense to create a seed file in order to quickly populate the `categories` table with several records. After all, I needed categories for development and testing purposes, and it made sense to include these same categories (`Laravel`, `CakePHP`, `Books`, `Conferences`, etc.) in the production database, meaning having a repeatable insertion solution on hand would be most convenient.

Laravel supports database seed files, storing them inside `database/migrations`. An Artisan seed file generator isn't yet available, so you'll have to manually create a file in this directory in order to take advantage of seeding. Inside the PHPLeaks' source you'll find the category table seeder in `database/seeds/CategoryTableSeeder.php`. Here's a partial recreation of the file:

```
<?php

use Illuminate\Database\Seeder;
use Illuminate\Database\Eloquent\Model;
use Phpleak\Category;

class CategoryTableSeeder extends Seeder {

  public function run()
  {

    Category::create([
      'name' => 'Laravel'
    ]);

    Category::create([
      'name' => 'Zend Framework'
    ]);

    Category::create([
      'name' => 'Conferences'
    ]);

    Category::create([
      'name' => 'Books'
    ]);

    ...

  }

}
```

With the seed file in place all you need to do is add it to the `DatabaseSeeder.php` file, which is responsible for executing the seeding process. You'll reference the new seed file class name inside the `run()` method like so:

```
public function run()
{
    Model::unguard();

    $this->call('CategoryTableSeeder');
}
```

If you plan on running the seeder several times during the development process (likely so), you'll want to clean out the table before populating it anew. You can do so by retrieving the table and using the `delete` method:

```
public function run()
{
    Model::unguard();

    DB::table('categories')->delete();
    $this->call('CategoryTableSeeder');

}
```

After updating `DatabaseSeeder.php` you'll want to run `composer dump-autoload` so Laravel is made aware of the new seed class. After doing so, run the following command to seed the database:

```
$ php artisan db:seed
```

Create the Categories Controller

With the `Category` model and associated table in place, I next created the `Category` controller:

```
$ php artisan make:controller CategoryController
Controller created successfully.
```

This created a resourceful controller containing the seven standard actions (`index`, `show`, `create`, `store`, `edit`, `update`, and `destroy`), however I only planned on using two of the actions (`index` and `show`) and so restricted the controller routing to just those two actions in the `routes.php` file:

```
Route::resource('category', 'CategoryController',
    ['only' => ['index', 'show']]);
```

These actions and associated views are pretty straightforward so I won't at this time dive into any additional detail regarding their implementation. Of course, have a look at the code and e-mail me with any questions.

With the category-related model and controller in place, it's time to implement the real core feature of PHPLeaks: link addition and display!

Integrating Links

The vast majority of content found on PHPLeaks pertains to links submitted by myself or other registered members. Each link includes a title, URL and description, and is associated with a category and the submitting user. Further, link "popularity" is determined by tracking the number of times users favorite and follow them.

Of course, none of these features can be implemented without first creating a `Link` model, so let's do that now:

```
$ php artisan make:model Link
Model created successfully.
Created Migration: 2015_03_27_195343_create_links_table
```

Next, open the newly created migration file and modify it like so:

```
public function up()
{
    Schema::create('links', function(Blueprint $table)
    {
        $table->increments('id');
        $table->integer('category_id')->unsigned();
        $table->foreign('category_id')
              ->references('id')->on('categories')
              ->onDelete('cascade');
        $table->string('name');
        $table->string('url');
        $table->string('description');
        $table->timestamps();
    });
}
```

In order to formalize this relationship inside Laravel you'll need to update the `Link` and `Category` models after running the above migration. We'll do that next.

Associating the Category and Link Models

In the previous migration snippet I placed special emphasis on the lines which pertain to how a category and link are related. Each link *belongs to* a category, meaning a record from the `categories` table is identified as a foreign key in the `links` table. This means a category *has many* links, because any given category's ID can appear multiple times in the `link` table. To inform Laravel of this relationship you'll need to update the `Link` and `Category` models. Begin by opening the `Link` model and adding a `category` method to it which allows you to easily retrieve a link's category:

```
class Link extends Model {

    ...

    public function category()
    {
      return $this->belongsTo('Phpleaks\Category');
    }

}
```

Next, open the `Category` model and add a `links` method which allows you to easily retrieve a category's links:

```
class Category extends Model {

    ...

    public function links()
    {
        return $this->hasMany('Phpleaks\Link');
    }

}
```

With these two associations defined you can do all sorts of interesting things, such as retrieve the category name of a given link:

```
$link = Link::find(45);
$category = $link->category->name;
```

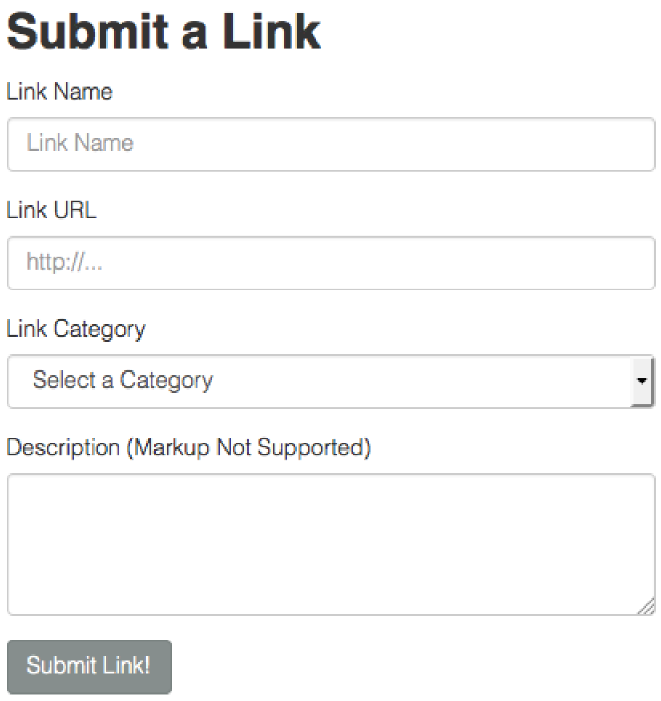
You can also iterate over a category's associated links:

```
$category = Category::find(12);
foreach ($category->links as $link)
{
    echo $link->name;
}
```

With these associations in place let's talk about how the link submission form was implemented as this feature demonstrates numerous very useful concepts.

Implementing a Link Submission Form

The link submission form (depicted in the below screenshot) is available to all registered users, allowing them to add a link to the site by providing a few key pieces of information, including a link name, URL, category, and description.



The PHPLeaks link submission form

This form is served through the `Link` controller, which is just a standard resourceful controller:

```
$ php artisan make:controller LinkController
Controller created successfully.
```

If you open the PHPLeaks `LinkController.php` file, you can see how the `create` and `store` actions are implemented. In this section I'll highlight some of what I consider to be the more confusing aspects of such a feature, beginning with how the form's `Link Category` select box is populated. The `Link` controller's `create` action looks like this:

```
public function create()
{
    $categories = \DB::table('categories')->lists('name', 'id');
    return view('link.create')->with('categories', $categories);
}
```

Thanks to the query builder's `lists` method, the `$categories` variable contains an *array* of category IDs and names. This is useful because a select box requires two values for each select option: a label and value. This variable is passed into the view, and subsequently passed into the `Form::select` helper, as demonstrated here:

```
<div class="form-group">
    {!! Form::label('Link Category') !!}<br />
    {!! Form::select('category',
        (['0' => 'Select a Category'] + $categories),
            null,
            ['class' => 'form-control']) !!}
</div>
```

When the helper is rendered to the browser the resulting HTML looks like this:

```
<select name="category" class="form-control">
    <option value="0">Select a Category</option>
    <option value="1">Laravel</option>
    <option value="2">Zend Framework</option>
    <option value="3">CakePHP</option>
    <option value="4">PHP</option>
</select>
```

So how is this category associated with the controller? I'll talk about this next.

Validating and Saving the Form Data

The link insertion form is submitted to the `Link` controller's `store` action per the usual RESTful conventions. However, thanks to Laravel 5's new form requests feature, the data is first passed through a validator located inside the form request associated with this particular form. The form request associated with the link creation form is called `LinkCreateFormRequest` and you'll find it inside `app/Http/Requests/LinkCreateFormRequest.php`. It was created using the following command:

```
$ php artisan make:request LinkCreateFormRequest
```

A simplified version of the `LinkCreateFormRequest.php` file is presented next:

```
<?php namespace Phpleaks\Http\Requests;

use Phpleaks\Http\Requests\Request;

class LinkCreateFormRequest extends Request {

    public function authorize()
    {
        return true;
    }

    public function rules()
    {

        return [
          'name' => 'required',
          'url' => 'required|url|unique:links,url',
          'category' => 'required|integer|min:1',
          'description' => 'required|max:300'
        ];

    }

    public function messages()
    {
        return [
            'name.required' => 'Please provide a brief link description',
            'url.required' => 'Please provide a URL',
            'url.url' => 'A valid URL is required',
            'url.unique' => 'This URL has already been submitted',
            'category.required' => 'Please associate this link with a category',
            'category.min' => 'Please associate this link with a category',
            'description.required' => 'A description is required',
            'description.max' => 'The description can\'t be longer than 300 char\
acters'
        ];
    }

}
```

This form request contains three methods: `authorize()`, `rules()`, and `validate()`. I'll summarize the purpose of each here:

- The `authorize()` method: This method can be used to restrict access to the form processor. If you look in the actual `LinkCreateFormRequest.php` file you'll see that I determine whether the submitting user is authenticated.
- The `rules()` method: This method defines the validation rules associated with each form field. The array keys must match the field names, in this case `name`, `url`, `category`, and `description`. Each key's associated value defines the validation rules used in conjunction with the form field.
- The `messages()` method: This method defines custom validation error messages which will be presented to the user in case validation fails. Note this method is not included in the form request by default; you'll need to add it. If you don't add it then the default validation error messages will be used instead.

With the form request defined you'll pass it into the `store` method as a parameter. The great aspect of form requests is that this is the *only* step you need to take in order to ensure the form request is used. If validation passes then you can use the `$request` object inside the `store` method. If validation fails then the code found inside the `store` action never even executes! Here's a simplified version of the `Link` controller's `store` action:

```php
use Phpleaks\Http\Requests\LinkCreateFormRequest;

...

public function store(LinkCreateFormRequest $request)
{

    $link = new Link(array(
      'name' => $request->get('name'),
      'url' => $request->get('url'),
      'description' => $request->get('description')
    ));

    $link->category()->associate(
        Category::find($request->get('category'))
    );

    $link->save();

    return \Redirect::route('link.show',
        array($link->id))->with('message',
        'Your link has been added!');

}
```

I'd like to particularly highlight these lines:

```
    $link->category()->associate(
        Category::find($request->get('category'))
    );
```

If you recall from the earlier discussion pertaining to the category select box found in the link submission form, the category ID will be passed along as the category parameter. This means you can find the category using this ID and then pass that object into the `associate()` method as demonstrated here.

Restricting Access to Registered Users

In order to at least hinder opportunities to spam the site with irrelevant links, users are required to register before they can submit a link. While I'll talk about various user registration and profile features in the next chapter I nonetheless thought it wise to discuss how this restriction was accomplished here.

The link form is protected in two different fashions. First, I configured the `Link` controller's class constructor to grant only authorized users access to the `create` and `store` actions:

```
class LinkController extends Controller {

    public function __construct() {
        $this->middleware('auth',
            ['only' => ['create', 'store']]);
    }

    ...

}
```

The `auth` middleware is included in every new Laravel 5 application. You can learn more about how it works by having a look at `app/Http/Middleware/Authenticate.php`.

In addition to the middleware restriction, I've updated the `LinkCreateFormRequest.php` form request to determine whether the user is authenticated in the `authorize()` method:

```
public function authorize()
{
    if (\Auth::check()) {
        return true;
    } else {
        return false;
    }
}
```

Frankly this is probably overkill since the middleware restriction is also associated with the `store` method, but it doesn't hurt to include it if for any other reason that another developer looking at your code will immediately know this particular form request is intended only for use in conjunction with authenticated users.

Using Sluggable Category and Link URLs

Frameworks such as Laravel and CakePHP do a great job of creating user-friendly URLs by default, meaning the days of creating applications sporting ugly URLs like this are long gone:

```
http://phpleaks.com/category.php?cat=12
```

Instead, Laravel will transform a URL like the above into something much more readable, such as:

```
http://phpleaks.com/category/12
```

However, while an improvement this URL really isn't particular informative. After all, what does the `12` even mean to the user? You and I know it is an integer value representing the primary key of a record found in the `categories` table, but it would be much more practical to instead use a URL that looks like this:

```
http://phpleaks.com/category/laravel
```

This "friendly" parameter is known as a *slug*, and thanks to the eloquent-sluggable[30] package it's surprisingly easy to integrate sluggable URLs into your Laravel application. Begin by adding the eloquent-sluggable package to your `composer.json` file:

```
$ composer require cviebrock/eloquent-sluggable >=3.0.0-alpha
```

At the time of this writing I had to instead reference the master (`dev-master`) branch in order to workaround a conflict with the latest version of Laravel. Have a look at the project's `composer.json` file to see how this is done.

Next, add the `SluggableServiceProvider` to your `config/app.php` file's `providers` array:

[30]https://github.com/cviebrock/eloquent-sluggable

```
'providers' => [
    ...
    'Cviebrock\EloquentSluggable\SluggableServiceProvider',
    ...
],
```

After saving these changes, publish the package's configuration file by executing the following command:

```
$ php artisan vendor:publish
```

You'll find the configuration file in `config/sluggable.php`. While there's nothing you need to change in this fire in order to begin using the package, I nonetheless suggest having a look at it as there are several opportunities to override various default settings if necessary.

Creating Sluggable Models

With the eloquent-sluggable package installed you'll need to update your models and underlying tables to enable the sluggable feature. This is fortunately incredibly easy to do. For instance, to add slugs to the `Category` model, modify it like so:

```
use Cviebrock\EloquentSluggable\SluggableInterface;
use Cviebrock\EloquentSluggable\SluggableTrait;

class Category extends Model implements SluggableInterface {

    use SluggableTrait;

    protected $sluggable = array(
        'build_from' => 'name',
        'save_to'    => 'slug',
    );

    ...

}
```

There are several important changes to this model, including:

- The eloquent-sluggable interface and trait are referenced at the top of the file.
- The model continues to extend `Model` but additionally implements `SluggableInterface`.

- Inside the class body you'll see the model uses the `Sluggable` trait.
- An array-based property named `$sluggable` is defined which identifies the database column which should be used to create the slug and the database column to which the slug should be saved.

After saving the model changes you'll need to create a migration which adds the `slug` column to the table. You can do this manually or preferably use the following Artisan command eloquent-sluggable makes available to you:

```
$ php artisan sluggable:table categories
```

After running the migration, the eloquent-sluggable package will *automatically* create the slugs for you any time a new record is added to the `categories` table! Of course, if you're adding this capability to an existing table that already contains records then you'll need to update each record. One of the easiest ways to do this is by entering the `Tinker` console, selecting all records and then saving them back to the database after using the eloquent-sluggable's `resluggify` method:

```
$ php artisan tinker
Psy Shell v0.4.3 (PHP 5.5.21 â€" cli) by Justin Hileman
>>> namespace Phpleaks;
>>> $categories = Category::all();
>>> foreach ($categories as $category) {
... $category->resluggify();
... $category->save();
... }
>>>
```

With the slugs in place, all you need to do is retrieve the desired record by slug rather than the integer ID. The package provides a useful helper method for doing so called `findBySlug`. For instance to find the category record associated with the slug `laravel` you'll execute the following command:

```
$category = Category::findBySlug('laravel');
```

Of course, when working within an action you'll be passing along the slug as the ID so for instance if you have a look at the `Link` controller's `show` action you'll see the `$id` parameter is passed into the `findBySlug` method like so:

```php
public function show($id)
{
    $link = Link::findBySlug($id);
    return view('link.show')->with('link', $link);
}
```

As this section hopefully indicates, integrating sluggable URLs into your application is incredibly easy, and certainly improves the readability of your URLs!

Summary

Obviously PHPLeaks.com won't amount to much without rock-solid link addition, management and findability capabilities so clearly what's discussed in this chapter is just the start of what will surely be a lengthy evolutionary process. However hopefully this information at least will help set you off in the proper direction particularly if you plan on creating a project which involves multiple types of associations.

Chapter 4. Integrating User Accounts

PHPLeaks' success will ultimately rest upon the size and quality of the contributing users. In order to provide features such as link submission, favoriting and tracking, users must be identifiable by way of a registered account. Fortunately Laravel 5 is bundled with a really convenient user account integration features, however in most cases you'll want to make some slight modifications to the default implementation. In this chapter I'll touch upon the changes I made to accommodate the PHPLeaks project requirements.

Redefining the Default Root Route

In addition to the user account controllers found in `app/Http/Controllers/Auth`, each new Laravel 5 project includes the controllers `HomeController.php` and `WelcomeController.php`. Unauthenticated users attempting to access the application's so-called "home page" are routed to the `Welcome` controller, while authenticated users are routed to the `Home` controller. This is useful when you want to provide a great deal of customized content to authenticated users, however in many cases you'll want to route both authenticated and unauthenticated users to the same location. These controller routes are defined in the `routes.php` file like so:

```
Route::get('/', 'WelcomeController@index');

Route::get('home', 'HomeController@index');
```

As you can see, the `Welcome` controller is identified as being the destination when the root URI (`/`) is accessed, however a middleware named `guest` defined within the `Welcome` controller's constructor will cause authenticated users to be redirected to the `Home` controller:

```
public function __construct()
{
        $this->middleware('guest');
}
```

You can see how the `guest` middleware is implemented within `app/Http/Requests/RedirectIfAuthenticated.php` Because I want *all* users to land on the `Welcome` controller's `index` action, I simply commented out this middleware call.

Updating the Post-Authentication Route

Even given the above changes, users will still be redirected to the `Home` controller after successfully signing into the site. This is because `Home` is identified as the default post-authentication path inside the Laravel framework's `AuthenticatesAndRegistersUsers` class should no other path be identified. Fortunately you can easily override this default by adding the following property to the `Auth` controller (found in `app/Http/Controllers/Auth/AuthController.php`):

```
protected $redirectPath = '/';
```

With this in place, authenticated users will be redirected to the root URI following a successful signin.

Updating the Password Recovery E-mail

Presuming you've configured Laravel's mail delivery feature as discussed in Chapter 1, then your application will be able to e-mail users attempting to recover a lost password. This e-mail contains a one-time URL which when clicked will take the user to a page where he can reset the password. You'll probably want to update the text found in this e-mail to suit your particular needs. You can easily do so since the e-mail is found in the Blade file `resources/views/emails/password.blade.php`. Have a look at the file found in the PHPLeaks project for a simple example.

Summary

This is currently a very short chapter, simply because Laravel's default account registration and authentication capabilities work so well that for more applications there's very little to change! Even so there remains plenty to talk about pertaining to user accounts so stay tuned as I'll be sure to expand this chapter in the near future.

Chapter 5. Creating an Administration Console

Most web applications require at least some level of ongoing monitoring and maintenance. While there are several approaches one could take to manage the application, the most commonplace involves creating a restricted administration interface accessible only to trusted individuals. In this chapter I'll show you how I created a simple such interface for PHPLeaks.com.

Creating a Route Group

Laravel supports a great feature known as a *route group* which you can use to manage a set of routes with minimal redundancy. For instance, you might wish to create a set of controllers for managing submitted links and categories, and for viewing and potentially banning registered users. These controllers might be named `LinkController.php`, `CategoryController.php`, and `UserController.php`, but these names are already used by existing PHPLeaks.com controllers. Rather than devise some sort of contrived and clumsy controller names you can instead create a special namespace for these controllers (`admin` sounds like a fine name) and manage them in a separate directory. For instance you can create a namespaced controller by prefixing the controller name with the desired namespace designation like so:

```
$ php artisan make:controller admin/UserController
```

A new directory named `admin` will be created inside `app/Http/Controllers`, with the new `UserController.php` controller file found inside `admin`. Next, open `routes.php` and add the following route definition:

```
Route::group(['prefix' => 'admin', 'namespace' => 'admin'], function()
{
    Route::resource('user', 'UserController');
});
```

This creates a route group, informing Laravel that all routes defined inside the group use the namespace `admin`. Additionally, the routes are accessible by *prefixing* the RESTful controller's typical URI naming convention with `admin`. You're not required to use an identical prefix and namespace; for instance the prefix could just as easily be called `panel` if you please.

With the route group defined, open the `admin/UserController.php` controller and modify the `index` action to look like this:

```
public function index()
{
        $users = User::orderBy('created_at', 'desc')->paginate(25);
        return view('admin.user.index')->with('users', $users);
}
```

Finally, create a new directory named `admin` inside `resources/views`. You'll use this directory to house any views associated with the namespaced controllers. Next, create a directory inside `admin` named `user`. Create a file inside the `user` directory named `index.blade.php`. This will serve as the view for the `admin/UserController.php`'s `index` action. Add the following contents to it:

```
@extends('layouts.master')

@section('content')

<h1>Registered Users</h1>

@if ($users->count() > 0)

        <table class="table table-striped">
                <thead>
                        <tr>
                        <th>Name</th><th>E-mail</th>
                        </tr>
                </thead>
                <tbody>
                        @foreach ($users as $user)
                                <tr>
                                <td>{{ $user->name }}</td>
                                <td>{{ $user->email }}</td>
                                </tr>
                        @endforeach
                </tbody>
        </table>

        {!! $users->render() !!}

@else

        <p>
        No users registered.
        </p>
```

```

@endif

@endsection
```

The contents of this controller action and view are all pretty standard so I won't belabor the implementation. However it is worth pointing out that while I extended the standard application layout, you might consider creating a custom streamlined administration layout that incorporates Bootstrap and the application styles but otherwise removes any unneeded widgets and other information.

With these various items in place, navigate to `/admin/user` and you should see a paginated list of registered users! From here you can complete the `UserController` actions much in the same manner you would any other RESTful controller, creating forms for editing users and even adding custom attributes for banning users who are spamming the community.

Restricting Administration Access

The administrative console provides a great solution for easily monitoring and managing users and site content, however there remains one significant issue: the controllers are currently open to anybody who happens to access the correct URL! We'll want to restrict access only to those registered users who are identified as administrators. There are several ways one could go about implementing access control of this sort, however if the only requirement is that users be separated into two groups (administrators and non-administrators), you can identify administrators simply by creating and setting an `is_admin` flag in the `users` table. After creating and setting this flag, you can write a simple middleware for determining whether a user is an administrator, and then grant or deny access to the administration console accordingly.

Begin by creating a migration which will add the `is_admin` flag to the `users` table:

```
$ php artisan make:migration add_is_admin_to_user_table --table=users
Created Migration: 2015_03_27_183649_add_is_admin_to_user_table
```

Next, open the newly created migration file and update the `up()` and `down()` methods to look like this:

```
public function up()
{
        Schema::table('users', function(Blueprint $table)
        {
                $table->boolean('is_admin')->default(false);
        });
}

public function down()
{
        Schema::table('users', function(Blueprint $table)
        {
                $table->dropColumn('is_admin');
        });
}
```

Save the changes and run the migration:

```
$ php artisan migrate
Migrated: 2015_03_27_183649_add_is_admin_to_user_table
```

With the `is_admin` column added, sign in to your database and update the specific record(s) associated with the users you would like to be treated as administrators. For instance when using the mysql client you'll execute the following command:

```
mysql> update users set is_admin = true where email='wj@wjgilmore.com';
```

With the administrators identified, let's create and integrate the middleware responsible for ensuring only administrators can access the restricted console.

Restricting the Administration Console with Middleware

Let's create the middleware which will be used to determine whether a user is indeed an administrator before granting access to the administration console:

```
$ php artisan make:middleware AdminAuthentication
Middleware created successfully.
```

Open the newly created middleware (`app/Http/Middleware/AdminAuthentication.php`) and modify the file to look like the following. To make it perfectly clear what has been added I've bolded the inserted lines:

```php
<?php namespace Phpleaks\Http\Middleware;

use Closure;
use Illuminate\Contracts\Auth\Guard;
use Illuminate\Http\RedirectResponse;

class AdminAuthentication {

        protected $auth;

        public function __construct(Guard $auth)
        {
                $this->auth = $auth;
        }

        public function handle($request, Closure $next)
        {
                if ($this->auth->check())
                {
                        if ($this->auth->user()->is_admin == true)
                        {
                                return $next($request);
                        }
                }

                return new RedirectResponse(url('/'));

        }

}
```

Next, open `app/Http/Kernel.php` and register the middleware inside the `$routeMiddleware` array:

```php
protected $routeMiddleware = [
        ...
        'admin' => 'Phpleaks\Http\Middleware\AdminAuthentication',
];
```

After saving these changes, only one step remains. You'll need to update the `admin` route group to use the middleware:

```
Route::group(
        ['prefix' => 'admin',
        'namespace' => 'admin',
        'middleware' => 'admin'], function()
        {
            Route::resource('user', 'UserController');
        });
```

With the route group updated, your administration console will be available only to users having the `is_admin` column enabled!

Summary

Creating a restricted administration console inside a Laravel application is surprisingly easy to accomplish once you understand the general approach. Of course, depending upon the nature of the actions that can be taken inside the console you might consider implementing additional access restriction measures such as preventing access outside of a specific IP address or block. Additionally you'll certainly want to enable SSL. In any case, hopefully this chapter made clear just how quickly such a solution can be implemented!