Beginning Laravel

Build Websites with Laravel 5.8

—

Second Edition

—

Sanjib Sinha

Apress®

Beginning Laravel

Build Websites with Laravel 5.8

Second Edition

Sanjib Sinha

Apress®

Beginning Laravel

Sanjib Sinha
Howrah, West Bengal, India

ISBN-13 (pbk): 978-1-4842-4990-1 ISBN-13 (electronic): 978-1-4842-4991-8
https://doi.org/10.1007/978-1-4842-4991-8

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Nikhil Karkal
Development Editor: Mathew Moodie
Coordinating Editor: Divya Modi

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-4990-1. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

This book is dedicated to Dr. Baidyanath Haldar,
who dedicated his life to eradicating leprosy in India.

You taught me to appreciate the value of lifelong learning.

Sir, I truly miss you.

Table of Contents

About the Author

Sanjib Sinha is a certified .NET Windows and web developer, specializing in Python, security programming, PHP, Java, and Dart. He won Microsoft's Community Contributor Award in 2011. As a published author, Sanjib Sinha has written *Beginning Ethical Hacking with Kali Linux*, *Beginning Ethical Hacking with Python*, and *Beginning Laravel* for Apress.

About the Technical Reviewer

Yogesh Sharma is a solution architect and full-stack engineer working at Mphasis Ltd. Primarily responsible for the design, development, configuration, and migration of various applications and modules with exceptional service quality over major platforms and service providers for DXC Technologies. He likes to cook for his family and spends free time fragging his friends on PlayStation. He can be reached at `https://www.linkedin.com/in/yogisharma24/`.

Acknowledgments

I wish to record my gratitude to my wife, Kaberi, for her unstinting support and encouragement in the preparation of this book.

I am extremely grateful to Mr. Matthew Moodie, lead development editor, for his numerous valuable suggestions, complimentary opinions, and thorough thumbing; to Nikhil Karkal, editor, and Divya Modi, coordinating editor; and to the whole Apress team for their persistent support and help.

In the preparation of this book, I have had to consult numerous open source documentation and textbooks on a variety of subjects related to PHP 7 and Laravel 5.8. I thank the countless authors who have written them. I hereby acknowledge my special indebtedness to the creator of Laravel, Mr. Taylor Otwell.

Introduction

After reading this book, you will be able to develop any web application using Laravel 5.8. It details all you need to know, including the Model-View-Controller pattern, SQLite databases, routing, Eloquent relations, authorization, middleware, events, broadcasting, APIs, and CRUD applications. I use four applications to explain various aspects of Laravel 5.8, and you can find the source code for all of them in this book's GitHub repository, which you can find at `http://www.apress.com/source-code`.

After showing how to set up your environment, the book explains routing, controllers, templates, and views in detail so that beginners can get started easily. It explains how a resourceful controller can become your great friend in keeping resources separated.

While introducing models, I also explain route model binding, model relations, and the relationship between models, databases, and Eloquent.

I also explain how user input and dependency injection work together and how Elixir and pagination work.

Understanding Eloquent relationships is important, so this book explains all types of relations, including one-to-one, one-to-many, many-to-many, has-many-through, and polymorphic.

While learning about Query Builder and Database Facade, you will also learn how you use fake database data for testing purposes.

Handling user data and redirecting it plays a big role in Laravel, so this is explained along with web form fundamentals and validations.

Artisan and Tinker are two great features of Laravel 5.8; you will learn about them in detail. You will also learn how to use SQLite in a small or medium-sized application.

In this book, you will get a detailed overview of authentication, authorization, and middleware; in addition, you will learn how authorization can be managed through Blade templates, gates, and policies.

After a detailed discussion of Laravel container and facades, you will learn how a mail template works, how one user can send a notification to another, and so on. Events and broadcasting also play vital roles in building complicated applications. The same is true for APIs; they are explained with examples.

At the end of the book, you will also get a glimpse of all the new features of Laravel 5.8.

CHAPTER 1

Introduction to Laravel

This book is intended for intermediate-level PHP developers. Advanced users may also find this book helpful to keep on hand as a quick reference. Either way, you should have a good grasp of object-oriented PHP7 programming knowledge to understand the examples.

Laravel's Flexibility

Laravel is a highly flexible PHP framework, and it makes complicated tasks easier to do.

Laravel provides powerful tools for making large, robust applications; however, to accomplish that, you need to understand a few key concepts such as how to use IoC containers or service containers to manage class dependencies. The expressive migration system is another concept that you will learn about in detail in Chapter 5.

Behind the scenes, Laravel uses many design patterns, and as you progress through this book, you will find that from the beginning Laravel helps you code in a loosely coupled environment. The framework focuses on designing classes to have a single responsibility, avoiding hard-coding in the higher-level modules, and allowing IoC containers to have abstractions that depend on the details. Higher-level modules do not depend on the lower-level modules, making coding a joyful experience.

As you progress through the book, you will find plenty of examples supporting this general paradigm. Because of this flexibility, Laravel has become one of the most popular frameworks today.

S. Sinha, *Beginning Laravel*, https://doi.org/10.1007/978-1-4842-4991-8_1

If you are not convinced, I will also say that without the help of a framework like Laravel, you would need to write thousands of lines of code that would probably start out by routing HTTP requests. In Laravel, it is simple and straightforward to route an HTTP request. This is defined in the `routes/web.php` file, as shown here:

```
//code 1.1
//Laravel Route Example
Route::get('/', function () {
    return view('welcome');
});
```

This defines the URL for an application. While defining the URL, Laravel also helps you return a Blade template page that you can use to nicely format and design your data. You will learn about this in detail in Chapter 3.

In any other framework, you won't know how complicated the requests will be and what your responses will be. So, you'll need response libraries to assist you in your framework, which could easily be a humongous task to implement.

Laravel can handle your request/response lifecycle quite easily. The complexity of requests does not matter here. Laravel's HTTP middleware consists of little HTTP layers that surround your application. So, every request must pass through these HTTP middleware layers before hitting your application. They can intercept and interrupt the HTTP request/response lifecycle, so you can stay calm about all the security matters that are most important. From the beginning, you have a well-guarded application interface when you install Laravel. You can think the first layer as the authentication layer, whereas the middleware tests whether the user is authenticated. If the user is not authenticated, the user is sent back to the login page. If the user passes the test, the user must face the second layer, which could consist of CSRF token validation. The process goes on like this, and the most common use cases of the Laravel middleware that protects your application are authentication, CSRF token validation, maintenance mode, and so on. When your application is in maintenance mode, a custom view is displayed for all requests.

For any experienced developer, it takes many months to build well-structured containers that can handle such complex requests and at the same time terminate the response lifecycle at the proper time.

Frameworks help you achieve this goal in a short period of time. Finding components is much easier; once you understand how routing works in Laravel, you can apply it quite easily for the rest of your working life.

In addition, Laravel's installation process is simple. With the help of the Composer dependency manager, you can manage it quite easily. In Chapter 2, I will discuss it in detail.

As a developer, you are here to solve multiple problems, and a framework like Laravel, in most cases, addresses those problems in an elegant way. So, you don't have to write tons of libraries and worry about version changes. The makers of Laravel have already thought about that.

Laravel's author, Taylor Otwell, summarizes the flexible features of Laravel as follows (from a PHPWorld conference in 2014):

- Aim for simplicity
- Minimal configuration
- Terse, memorable, expressive syntax
- Powerful infrastructure for modern PHP
- Great for beginners and advanced developers
- Embraces the PHP community

How Laravel Works

Laravel follows the MVC pattern of logic. But it has many more features that have enhanced the pattern's features, elevating them to a new level.

Laravel ships with a dependency injection (DI) mechanism by default (Figure 1-1).

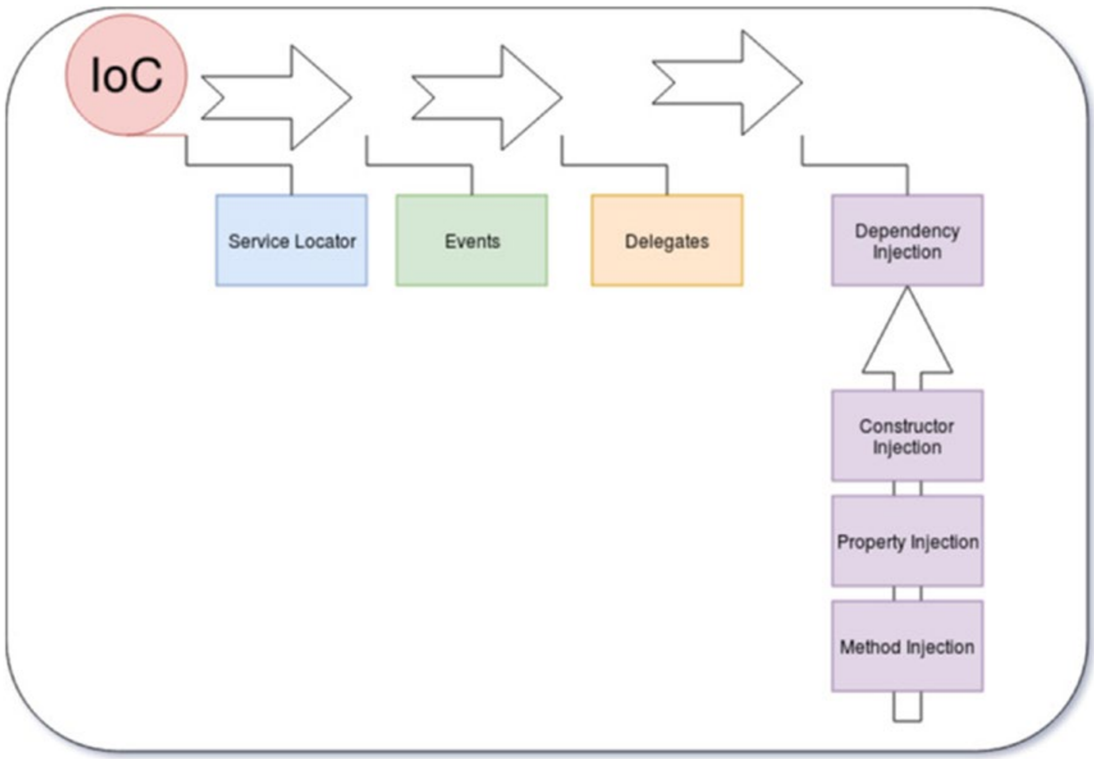


Figure 1-1. *How DI works with an IoC container*

If you are new to this important concept of software building and programming, don't worry. I will discuss it in great detail later. For now, know that the DI mechanism is a way to add a class instance to another class instance using a class constructor.

Let's consider a real-life example. Take a look at the following code first; then I will explain how IoC and DI work together in Laravel:

```
//How to use abstraction using higher-level and lower-level modules
interface PaymentInterface {
    public function notify();
}

class PaymentGateway implements PaymentInterface {

    public function notify() {

        return "You have paid through Stripe";
    }
}
class PaymentNotification {
    protected $notify;
    public function __construct(PaymentInterface $notify) {
        $this->notify = $notify;
    }
    public function notifyClient() {
        return "We have received your payment. {$this->notify->notify()}";
    }
}
$notify = new PaymentNotification(new PaymentGateway);
echo $notify->notifyClient();
```

With the previous code, you will get output like this:

```
We have received your payment, you have paid through Stripe.
```

The key concept of DI is that the abstraction (here, `PaymentInterface`) never worries about the details. The lower-level implementation modules (here, the `PaymentGateway` class) handle the details. In this case, it uses Stripe as a payment gateway. And the `PaymentNotification` class uses the abstraction to get those details. It also does not bother about what type of payment gateway has been used.

Now if the client changes the payment gateway from Stripe to PayPal, you don't have to hard-code it into the class that uses the payment gateway. In the implementation, you will change it, and any class into which you inject the implementation will reflect the change.

Depending on many classes is necessary. Managing those dependencies is not easy.

Using an IoC container, you can manage the class dependencies as I have shown in the previous code. The IoC container resolves the dependencies easily. In fact, the IoC container is a central piece of the Laravel framework. This container can build complex objects for you.

Note DI is a form of IoC, which is why the container that manages this is called an *IoC container*.

Laravel ships with more than a dozen service providers, and they manage the IoC container bindings for the core Laravel framework components.

The system loops through all the providers since you don't use them all the time. Suppose you are using a form to send data; `HTMLServiceProvider` helps you with that. As shown in Figure 1-2, the kernel sends the request through the middleware to the controller, which I'll cover in the next section.

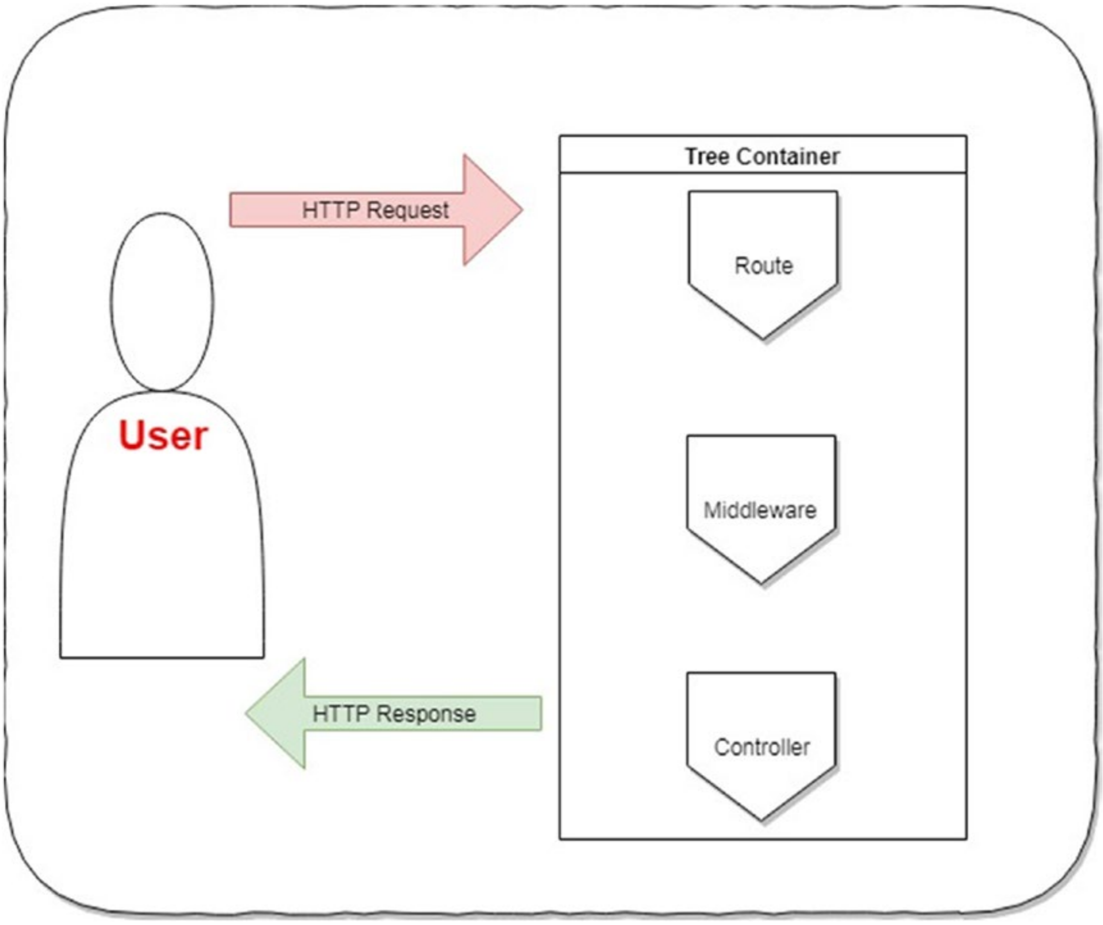


Figure 1-2. *How the route and middleware request and response cycle work together*

If you open the `config/app.php` file, you will find these lines of code:

```
//code 1.16
//config/app.php
'providers' => [

        /*
         * Laravel Framework Service Providers...
         */
        Illuminate\Auth\AuthServiceProvider::class,
        Illuminate\Broadcasting\BroadcastServiceProvider::class,
        Illuminate\Bus\BusServiceProvider::class,
        Illuminate\Cache\CacheServiceProvider::class,
...
App\Providers\EventServiceProvider::class,
        App\Providers\RouteServiceProvider::class,

    ],
```

This code is not shown here in its entirety, but this gives you an idea how the Laravel kernel system loops through those providers.

The Silex microframework, once maintained by Symfony, is now deprecated (at the time of writing this book), yet I encourage you to download it. You can study the similarities and differences compared to Laravel.

What Is the MVC Pattern?

How do you successfully implement user interfaces with Laravel and follow the separation of concerns principle? The Model-View-Controller (MVC) architecture is the answer. And Laravel helps you do this quite easily (Figure 1-3).

The workflow is simple: the user action reaches the controller from the view. The controller notifies the model. Accordingly, the model updates the controller. After that, the controller again updates the user.

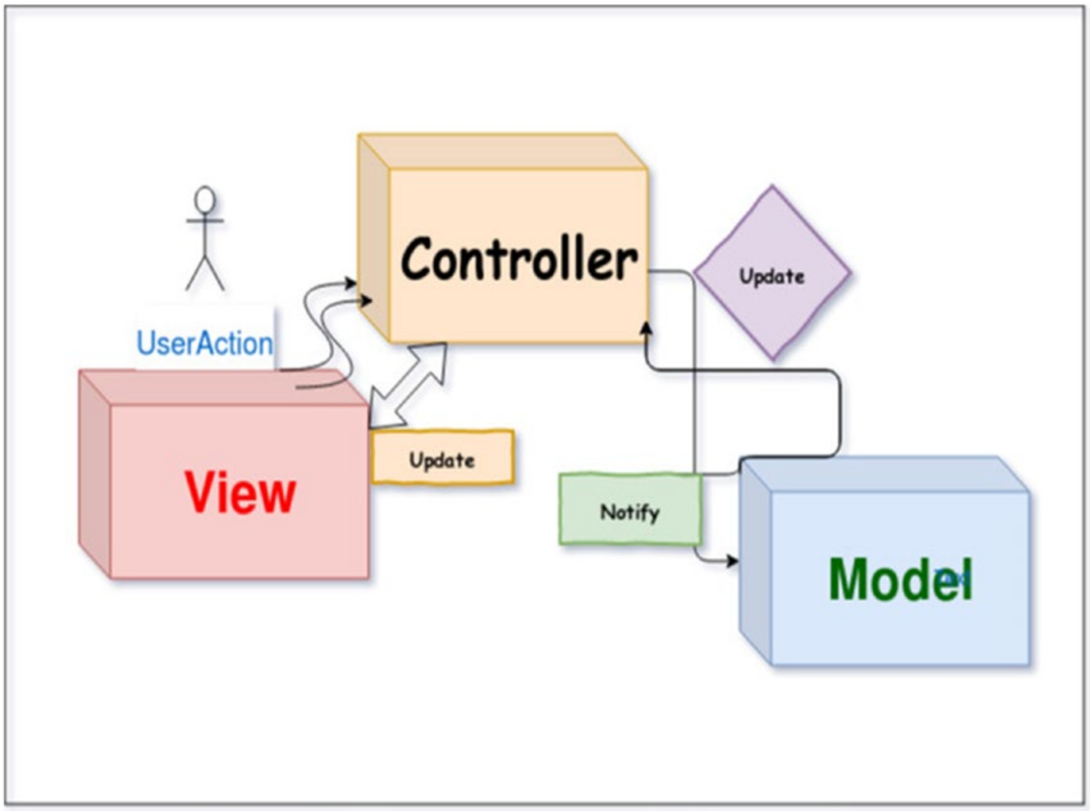


Figure 1-3. *How the MVC workflow works*

That is why it is commonly used and a popular choice. The separation of concerns principle usually separates the application logic from the presentation layers in Laravel. Quite naturally, your application will become more flexible, modular, and reusable.

Let's imagine a social networking application where you want to view a user's page. You may click a link that looks like this:

```
https://example.com/home/index/username
```

Here you can imagine that `home` is the controller, `index` is the method, and in the `username` section you can even pass an ID. This is pretty simple, and it takes you to the user's page. How you can make your app work through an MVC model like this?

As you might guess, you will get the user's data from a database. The model defines what data your application should contain. So, it is model's job to interact with the database. Now, a user's database will be constantly evolving; changes will take place, and the user will make updates. If the state of this data changes, the model will usually notify the view. If different logic is needed to control the view, the model notifies the controller.

Keeping the social media app in mind, the model would specify what data a list item should contain, such as first name, last name, location, and so on.

The role of the controller is critical. It contains the programming logic that updates the model and sometimes the view in response to input from the users of the app. In other social media apps, almost the same thing happens. The app could have input forms to allow you to post or edit or delete the data. The actions require the model to be updated, so the input is sent to the controller, which then acts upon the model, and the model then sends the updated data to the view. Now, this view can be manipulated in a different manner; you can use your layout template engine. In some cases, the controller can handle these tasks independently without needing to update the model.

Compared to the functions of the model and the controller, the workflow of the view consists of simple tasks. It will define only how the list is presented to another user. It will also receive the data from the model to display.

Actually, you have seen this pattern before. The data model uses some kind of database, probably MySQL in the LAMP technology. In such cases, your controller's code is written in PHP, and in your view part, you have used a combination of HTML and CSS.

How the MVC Pattern Works

Imagine a portal, where your user interacts with the user interface in some way, such as by clicking a link or submitting a form. Now, the controller takes charge and handles the user input event from the user interface. Consequently, the controller notifies the model of the user action. Usually, the state of model changes. Suppose the controller updates the status of the user. It interacts with the model, and the model has no direct knowledge of the view. The model passes data objects to the controller, and then your controller generates the content with that dynamic data.

Because of its simple iterations and the principle of separation of concerns, the MVC pattern is often found in web application.

Now, the MVC pattern can be interpreted in different ways: A section of people thinks that the actual processing part is handled by the model, and the controller handles only the input data. In such interpretations, the input-processing-output flow is represented by Controller-Model-View; here the controller interprets the mouse and keyboard inputs from the user.

Therefore, a controller is responsible for mapping end-user actions to application responses, whereas a model's actions include activating business processes or changing the state of the model. The user's interactions and model's response decide how a controller will respond by selecting an appropriate view.

To summarize, an MVC pattern must have a minimum of three components, each of which performs its own responsibilities. Now many MVC frameworks add extra functionalities such as a data access object (DAO) to communicate with the relational database.

In normal circumstances, the data flow between each of these components carries out its designated tasks; however, it is up to you to decide how this data flow is to be implemented. With the MVC framework, you will learn that the data is pushed by the controller into the view. At the same time, the controller keeps manipulating the data in the model. Now the question is, who is doing the real processing? The model or controller? Does the controller play an intermediary role, and behind the scenes, the model actually pulls the strings? Or, is the controller the actual processing unit controlling the database manipulations and representation simultaneously?

It does not matter as long as the data flows and the pattern works. In the Laravel applications in this book, you will implement MVC in a way so that the controller will do processing jobs, controlling the model and the view simultaneously.

When using the MVC framework, you must keep a separate DAO, and the model will handle the DAO as appropriate. Therefore, all SQL statements can be generated and executed in only one place. Data validation and manipulation can be applied in one place: the model. The controller will handle the request/response lifecycle, taking the input from the user interface and updating model accordingly. When the controller gets the green signal from the model, it sends the response to the view. The view is aware of only one thing: the display.

CHAPTER 2

Setting Up Your Environment

This chapter will help you install Composer, both locally and globally, and introduce you to Valet, Homestead, VirtualBox, Vagrant, and Forge. You will also learn how to create a new project in Laravel 5.8 using your Homestead development environment.

Composer

Composer is a dependency management tool in PHP. For any PHP project, you need to use a library of code. Composer easily manages this task on your behalf, helping you to declare those libraries and packages. You can also install or update any code in your library through Composer. Please visit `https://getcomposer.org` for more details.

Without the help of the Composer dependency manager, you cannot install Laravel. Moreover, as you dig deeper into Laravel, you will need even more packages and libraries to build your application.

You can install Composer in two ways.

- *Locally*: You can download and install Composer each time for every project rather than running Composer globally on your system. Because you install Composer locally each time, you will have no trace of the Composer package manager on your host machine.
- *Globally*: The global option is always preferable because once Composer is installed in your system's `bin` folder, you can call it anytime for any project. Whether you use Windows, Debian-based Linux like Ubuntu, or macOS, the global option always rules over the local one.

I'll show how to install Composer globally in the next section and then cover the local option.

S. Sinha, *Beginning Laravel*, https://doi.org/10.1007/978-1-4842-4991-8_2

Installing Composer Globally

For any Debian-based Linux distribution like Ubuntu or macOS, you can download Laravel using Composer in any folder anywhere and start working with it. If you use Windows, I suggest you download helper software like XAMPP that comes with the LAMP technology.

Installing Composer globally on your system requires a few steps, as shown here, in your terminal:

```
//how to install Composer globally
$ sudo apt-get update

$ sudo apt-get install curl php-cli php-mbstring git unzip

$ curl -sS https://getcomposer.org/installer -o composer-setup.php
```

Here you need to run a short PHP script to match the secret key. Issue this command on your terminal:

```
php -r "if (hash_file('SHA384', 'composer-setup.php') ===
'669656bab3166a7aff8a7506b8cb2d1c292f042046c5a994c43155c0be6190fa0355160742
ab2e1c88d40d5be660b410') { echo 'Installer verified'; } else { echo
'Installer corrupt'; unlink('composer-setup.php'); } echo PHP_EOL;"
```

You will get output like this:

```
Installer verified
```

After that, you need to move Composer to your `/usr/local/bin` folder. Run this command:

```
$ sudo php composer-setup.php --install-dir=/usr/local/bin
--filename=composer
```

You will get output like this:

```
All settings correct for using Composer
Downloading 1.1.1...

Composer successfully installed to: /usr/local/bin/composer
Use it: php /usr/local/bin/composer
```

Installing Laravel Globally

Next, you need to be using a LAMP server properly. This means you already have an operating system based on Linux, and you have the Apache web server, the MySQL database, and at least PHP 7.0. If this architecture is all set up, you can go ahead and download Laravel with a simple command like this:

```
//code 2.1
//installing Laravel with global composer
composer create-project --prefer-dist laravel/laravel
YourFirstLaravelProject
```

You can even be choosy with a particular Laravel version such as this:

```
//code 2.2
composer create-project --prefer-dist laravel/laravel blog "5.7.*"
```

Once you have Composer installed on your Windows system, you can use the same command to install Laravel in the `C:\xampp\htdocs\Dashboard\YourPreferredFolder` directory. You can use a terminal almost the same way as in Linux or macOS. There are small differences between operating systems; for example, on Linux or macOS, you use `ls -la` for a long listing, but in Windows, you use `dir`. These commands will tell you what is inside the Laravel folder.

On all operating systems, the next command to run Laravel is as follows:

```
//code 2.3
//running Laravel
$ php artisan serve
```

You can now go to `http://localhost:8000` to see what's there.

The `artisan` command is the most powerful tool Laravel comes with. With the help of the `artisan` command, you can create all the necessary components you need for your application.

The `artisan` command creates the following:

- Controller
- Models
- Migrations

- Seeds
- Tests

And there are many other components the `artisan` command creates. `artisan` also helps you work with your database using tools like Tinker. I will cover those features in later chapters.

You already saw how `artisan` helps you start the web server.

```
//code 2.4
//starting local web server in laravel
$ php artisan serve
```

You can also take the application into maintenance mode.

```
//code 2.5
//Taking application to the maintenance mode
$ php artisan down
```

After your maintenance work has been finished, you can issue the `up` command in the same way and again start the web server.

Clearing the cache is also simple.

```
//code 2.6
//Clearing cache
$ php artisan cache:clear
```

These are a few examples of how Laravel makes a developer's life easier. As you progress in the book, you will see `artisan` used frequently.

Installing Laravel Locally with Composer

In Ubuntu-like operating systems, for any local PHP project, you use the `/var/www/html` folder. In this section, you'll learn how to install a Laravel project there using Composer locally.

Installing Laravel locally with the help of Composer means you have to use the following command for each installation. You won't have Composer on your system globally like in the previous section. You can compare the following steps for installing locally with the global option I showed earlier; you'll see that the global option is easier.

You need to create a folder and name it `MyFirstLaravelProject`. Open your Ubuntu terminal (Control+Alt+T) and type the following command:

```
//code 2.7
$ cd /var/www/html
```

You can make a directory here with a simple command.

```
//code 2.8
$ sudo mkdir MyFirstLaravelProject
```

It will ask for your `root` user's password. Type the password, and a folder called `MyFirstLaravelProject` will be created.

Next, in this folder, you can download and install Composer locally. Then issue the following two commands one after another. First you type this:

```
//code 2.9
$ sudo php -r "copy('https://getcomposer.org/installer', 'composer-setup.
php');"
```

Next you type this:

```
//code 2.10
$ sudo php composer-setup.php
```

This will organize your Composer setup file to go further. Actually, Composer is ready to download packages for your upcoming projects. You can test it by creating a `composer.json` file in your `MyFirstLaravelProject` folder. In that `composer.json` file, type this:

```
//code 2.11
{
    "require": {
        "monolog/monolog": "1.0.*"
    }
}
```

Now you will learn how to install the `monolog` PHP package for your Laravel project. You're actually testing your Composer installer to see how it works. Type this command on your terminal:

```
//code 2.12
$ php composer.phar install
```

It'll take a moment to install the `monolog` package.

After the installation is complete, you'll find a `vendor` folder and a few Composer files in your project. Feel free to take a look at what is inside the `vendor` folder. There you'll find two folders: `composer` and `monolog`. Again, you can take a look at what they have inside them.

The time has come to install the latest version of Laravel through Composer. You can install Laravel just like the `monolog` package. This means you can write that instruction in your `composer.json` file and just update Composer. Open your terminal and write the following:

```
//code 2.13
$ sudo composer create-project --prefer-dist laravel/laravel blog
```

This will install the latest version of Laravel in the folder `blog` in your Laravel project, named `MyFirstLaravelProject`. The first step is completed: you've installed Laravel in the `/var/www/html/MyFirstLaravelProject/blog` folder. Now you can go in that folder and issue the Linux command `ls -la` to see what is inside. You can also type the `php artisan serve` command to run your first Laravel application so that you can go to `http://localhost:8000` to see the welcome page. Remember, this installation has been done locally.

Introduction to Homestead, Valet, and Forge

In this section, the primary focus is on Homestead. I'll also cover VirtualBox and Vagrant because they work together with Homestead. In normal circumstances, Homestead represents an environment, and VirtualBox runs that environment. Vagrant, as part of that, helps to run the terminal, where you can create your Laravel project.

Homestead offers an entire Ubuntu virtual machine with automated Nginx configuration. The following list shows what you get in Homestead:

- - Ubuntu 18.04
- - Git
- - PHP 7.3
- - PHP 7.2
- - PHP 7.1
- - PHP 7.0
- - PHP 5.6
- - Nginx
- - Apache (optional)
- - MySQL
- - MariaDB (optional)
- - Sqlite3
- - PostgreSQL
- - Composer
- - Node (with Yarn, Bower, Grunt, and Gulp)
- - Redis
- - Memcached
- - Beanstalkd
- - Mailhog
- - Neo4j (optional)
- - MongoDB (optional)
- - Elasticsearch (optional)
- - ngrok
- - wp-cli

- - Zend Z-Ray
- - Go
- - Minio

If you want a fully virtualized Laravel development environment, then Homestead is the only answer. Moreover, Homestead is extremely flexible. You can use MySQL in the beginning, and later you can use MariaDB, MongoDB, PostgreSQL, or any database of your choice. If you feel comfortable with PHP, then you can use that too. In essence, Homestead is a far more superior Laravel development environment than any other option available.

Note that another option for Mac users is Valet. When doing local development, Homestead and Valet differ mostly in size and flexibility. Valet only supports the PHP and MySQL combination, although it is lightning fast. In the case of Valet, you need to install PHP and MySQL on your home Mac machine or you won't get a fully virtualized environment for developing Laravel applications.

So, my suggestion is to go for Homestead to take advantage of the easy local development environment.

Finally, Forge is a commercial service through which you can provision and deploy various PHP applications including Laravel. By using Forge, you can deploy your Laravel application to cloud services such as DigitalOcean, Linode, AWS, and more.

Forge: Painless PHP Servers

You can imagine Forge as a service like Homestead; however, you need to buy it. Homestead is a local development environment, but Forge can manage cloud servers so that you can focus on building your application.

On the Forge server, you can install any application like WordPress, Symfony, and of course Laravel. You can choose any cloud server, and the Laravel Forge product make your life easier when deploying your application on the cloud.

As mentioned, Forge is a commercial product and comes with all the facilities that a commercial product usually offers, such as SSL certificates, subdomain management, queue worker management and cron jobs, load balancing, and so on.

Just like the Homestead local development environment, Forge gives you the same cutting-edge server configuration so that you can take advantage of Ubuntu 18.04 LTS and tailored server configurations, complete with Nginx, PHP 7, MySQL, Postgres, Redis, Memcached, and automated security updates.

Deploying code is also easy as you can just push your master to your GitHub, Bitbucket, or custom Git repository and Forge will take it from there.

Installing VirtualBox and Vagrant

The Homestead virtualized development environment requires two things. First, you need a virtual machine, which can be any of the following:

- VMware (`https://www.vmware.com`)
- VirtualBox 5.2 (`https://www.virtualbox.org/wiki/Downloads`)
- Parallels (`https://www.parallels.com/products/desktop/`)
- Hyper-V (`https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/quick-start/enable-hyper-v`)

Second, after installing any one of these virtual machines, you need to install Vagrant.

```
https://www.vagrantup.com/downloads.html
```

Basically, Homestead will use any of these virtual machines as well as Vagrant to run the local server, creating a development environment for your Laravel application. The biggest advantage is you don't require any LAMP technology directly on your host machine anymore. Homestead is the best option not only for the Laravel application but for any PHP application. You can test any type of PHP application or any other framework in your Homestead development environment. That is the biggest advantage of Homestead.

You can add as many projects as you want to your Homestead development environment and test them locally.

Let's now go through the steps to install Homestead so you can see how to start your first Laravel application. The first step is to install a virtual machine. My choice is VirtualBox 5.2.

Note VMware is not free, although it is faster than others. Parallels needs extra plugins. Hyper-V is fine, but VirtualBox is easier to install and maintain.

I don't want you to waste time researching virtualized machines. You are here to learn Laravel, and therefore you need a free, easily installable virtual machine. Let's get started.

```
//code 2.14
// installing VirtualBox on MAC/Linux
$ sudo apt-get install virtualbox
```

For Windows, download the required EXE file from https://www.virtualbox.org/wiki/Downloads according to your Windows version and install it.

Next you need to install Vagrant. For macOS/Linux, the command is the same.

```
//code 2.15
// installing Vagrant
ss@ss-H81M-S1:~$ sudo apt-get install vagrant
[sudo] password for ss:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages were automatically installed and are no longer
required:
  gir1.2-keybinder-3.0 jsonlint libcdaudio1 libenca0 libjs-excanvas
  libkeybinder-3.0-0 libllvm5.0 libllvm5.0:i386 libmcrypt4
  libp11-kit-gnome-keyring:i386 libslv2-9 libsodium18 libvpx3:i386
  mercurial
  mercurial-common php-cli-prompt php-composer-semver
  php-composer-spdx-licenses php-json-schema php-symfony-console
  php-symfony-filesystem php-symfony-finder php-symfony-process
Use 'sudo apt autoremove' to remove them.
The following additional packages will be installed:
  bsdtar bundler fonts-lato libgmp-dev libgmpxx4ldbl libruby2.3 rake ruby
  ruby-bundler ruby-childprocess ruby-dev ruby-did-you-mean ruby-domain-
  name
  ruby-erubis ruby-ffi ruby-http-cookie ruby-i18n ruby-listen ruby-log4r
  ruby-mime-types ruby-minitest ruby-molinillo ruby-net-http-persistent
  ruby-net-scp ruby-net-sftp ruby-net-ssh ruby-net-telnet ruby-netrc
  ruby-nokogiri ruby-power-assert ruby-rb-inotify ruby-rest-client
  ruby-sqlite3 ruby-test-unit ruby-thor ruby-unf ruby-unf-ext ruby2.3
  ruby2.3-dev rubygems-integration sqlite3
Suggested packages:
  bsdcpio gmp-doc libgmp10-doc libmpfr-dev ri publicsuffix sqlite3-doc
The following NEW packages will be installed:
  bsdtar bundler fonts-lato libgmp-dev libgmpxx4ldbl libruby2.3 rake ruby
  ruby-bundler ruby-childprocess ruby-dev ruby-did-you-mean ruby-domain-name
  ruby-erubis ruby-ffi ruby-http-cookie ruby-i18n ruby-listen ruby-log4r
  ruby-mime-types ruby-minitest ruby-molinillo ruby-net-http-persistent
  ruby-net-scp ruby-net-sftp ruby-net-ssh ruby-net-telnet ruby-netrc
  ruby-nokogiri ruby-power-assert ruby-rb-inotify ruby-rest-client
  ruby-sqlite3 ruby-test-unit ruby-thor ruby-unf ruby-unf-ext ruby2.3
  ruby2.3-dev rubygems-integration sqlite3 vagrant
0 upgraded, 42 newly installed, 0 to remove and 5 not upgraded.
Need to get 9,248 kB of archives.
After this operation, 44.8 MB of additional disk space will be used.
Do you want to continue? [Y/n] y
Get:1 http://in.archive.ubuntu.com/ubuntu xenial/main amd64 fonts-lato all
2.0-1 [2,693 kB]
Get:2 http://in.archive.ubuntu.com/ubuntu xenial-updates/universe amd64
bsdtar amd64 3.1.2-11ubuntu0.16.04.4 [48.0 kB]
Get:3 http://in.archive.ubuntu.com/ubuntu xenial/main amd64 rubygems-
integration all 1.10 [4,966 B]
Get:4 http://in.archive.ubuntu.com/ubuntu xenial/main amd64 rake all
10.5.0-2 [48.2 kB]
....
```

The code and output are not shown in full here for brevity.

After installing Vagrant, you can issue a single command to find out Vagrant's version.

```
//code 2.16
$ vagrant -v
```

```
//output
Vagrant 2.2.3
```

Be it VirtualBox or Vagrant, all of these software packages provide easy-to-use visual installers for Windows. For macOS and Linux, you can take advantage of the terminal.

To initialize Vagrant, you need to issue these commands one after another:

```
//code 2.17
$ vagrant box add ubuntu/trusty64
==> box: Loading metadata for box 'ubuntu/trusty64'
    box: URL: https://vagrantcloud.com/ubuntu/trusty64
==> box: Adding box 'ubuntu/trusty64' (v20181218.1.0) for provider:
         virtualbox
    box: Downloading: https://vagrantcloud.com/ubuntu/boxes/trusty64/
         versions/20181218.1.0/providers/virtualbox.box
==> box: Successfully added box 'ubuntu/trusty64' (v20181218.1.0) for
         'virtualbox'!
```

```
//code 2.18
ss@ss-H81M-S1:~$ vagrant init ubuntu/trusty64
A `Vagrantfile` has been placed in this directory. You are now
ready to `vagrant up` your first virtual environment! Please read
the comments in the Vagrantfile as well as documentation on
`vagrantup.com` for more information on using Vagrant.
ss@ss-H81M-S1:~$ vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Importing base box 'ubuntu/trusty64'...
==> default: Matching MAC address for NAT networking...
==> default: Checking if box 'ubuntu/trusty64' is up to date...
==> default: Setting the name of the VM: ss_default_1547081879727_59147
==> default: Clearing any previously set forwarded ports...
==> default: Clearing any previously set network interfaces...
==> default: Preparing network interfaces based on configuration...
    default: Adapter 1: nat
==> default: Forwarding ports...
    default: 22 (guest) => 2222 (host) (adapter 1)
==> default: Booting VM...
==> default: Waiting for machine to boot. This may take a few minutes...
    default: SSH address: 127.0.0.1:2222
    default: SSH username: vagrant
    default: SSH auth method: private key
    default:
    default: Vagrant insecure key detected. Vagrant will automatically
             replace
    default: this with a newly generated keypair for better security.
    default:
    default: Inserting generated public key within guest...
    default: Removing insecure key from the guest if it's present...
    default: Key inserted! Disconnecting and reconnecting using new SSH
             key...
==> default: Machine booted and ready!
==> default: Checking for guest additions in VM...
    default: The guest additions on this VM do not match the installed
             version of
    default: VirtualBox! In most cases this is fine, but in rare cases it can
    default: prevent things such as shared folders from working properly.
             If you see
    default: shared folder errors, please make sure the guest additions
             within the
    default: virtual machine match the version of VirtualBox you have
             installed on
    default: your host and reload your VM.
    default:
    default: Guest Additions Version: 4.3.36
    default: VirtualBox Version: 5.1
==> default: Mounting shared folders...
    default: /vagrant => /home/ss
...
```

The code isn't shown in full for brevity in the book. These commands basically have initialized Vagrant, and now you can also add the Composer packages. Vagrant has mounted the shared folders where you will keep your projects from now on. It will be in code in your home directory, like this:

```
//home/user/code/YourProjectsHere
```

In my case, since my username is ss, it looks like this:

```
/home/ss/code/
```

I will keep Laravel and my PHP projects here in the future, in different subfolders. The next command will help you to add Homestead through Vagrant:

```
//code 2.19
$ sudo composer global require "laravel/homestead=~2.0"
[sudo] password for ss:
Changed current directory to /home/ss/.composer
Do not run Composer as root/super user! See https://getcomposer.org/root
for details
./composer.json has been created
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 6 installs, 0 updates, 0 removals
  - Installing symfony/process (v3.4.21): Downloading (100%)
  - Installing psr/log (1.1.0): Downloading (100%)
  - Installing symfony/debug (v4.2.2): Downloading (100%)
  - Installing symfony/polyfill-mbstring (v1.10.0): Downloading (100%)
  - Installing symfony/console (v3.4.21): Downloading (100%)
  - Installing laravel/homestead (v2.2.2): Downloading (100%)
symfony/console suggests installing psr/log-implementation (For using the
console logger)
symfony/console suggests installing symfony/event-dispatcher ()
symfony/console suggests installing symfony/lock ()
Writing lock file
Generating autoload files
```

Installing Homestead Using Vagrant

The next step is the painless installation of Homestead so that you can get these virtualization processes completed. Let's install Homestead with the help of Vagrant, as shown here:

```
//code 2.20
// installing Homestead
$ vagrant box add laravel/homestead
==> box: Loading metadata for box 'laravel/homestead'
    box: URL: https://vagrantcloud.com/laravel/homestead
This box can work with multiple providers! The providers that it
can work with are listed below. Please review the list and choose
the provider you will be working with.

1) hyperv
2) parallels
3) virtualbox
4) vmware_desktop

Enter your choice: 3
==> box: Adding box 'laravel/homestead' (v6.4.0) for provider: virtualbox
    box: Downloading: https://vagrantcloud.com/laravel/boxes/homestead/
         versions/6.4.0/providers/virtualbox.box
    box: Download redirected to host: vagrantcloud-files-production.
         s3.amazonaws.com
==> box: Successfully added box 'laravel/homestead' (v6.4.0) for
         'virtualbox'!
...
```

The code is incomplete, although you can see that I have chosen number 3, as I have already installed VirtualBox and I am going to use VirtualBox for my further virtualization processes. It will take a few minutes to download the box, depending on your Internet connection speed.

Next, clone the repositories into a `Homestead` folder within your home directory.

```
//code 2.21
$ cd ~
$ git clone https://github.com/laravel/homestead.git Homestead
```

```
Cloning into 'Homestead'...
remote: Enumerating objects: 22, done.
remote: Counting objects: 100% (22/22), done.
remote: Compressing objects: 100% (16/16), done.
remote: Total 3232 (delta 14), reused 10 (delta 6), pack-reused 3210
Receiving objects: 100% (3232/3232), 689.62 KiB | 926.00 KiB/s, done.
Resolving deltas: 100% (1942/1942), done.
Checking connectivity... done.
```

Once you have cloned the Homestead repository, you should run the `bash init.sh` command from the Homestead directory to create the `Homestead.yaml` configuration file. The `Homestead.yaml` file will be placed in the `Homestead` directory. You need this file to edit your further connections.

```
//code 2.22
//for Mac and Linux...
$ bash init.sh
//for Windows...
 init.bat
```

Now you can check what your `Homestead` folder consists of, as shown here:

```
//code 2.23
$ cd ~/Homestead
$ ls -la
total 184
drwxrwxr-x  9 ss ss  4096 Jan 10 07:07 .
drwxr-xr-x 54 ss ss  4096 Jan 10 07:03 ..
-rw-rw-r--  1 ss ss   332 Jan 10 07:07 after.sh
-rw-rw-r--  1 ss ss  7669 Jan 10 07:07 aliases
drwxrwxr-x  2 ss ss  4096 Jan 10 07:03 bin
-rw-rw-r--  1 ss ss   187 Jan 10 07:03 CHANGELOG.md
-rw-rw-r--  1 ss ss   853 Jan 10 07:03 composer.json
-rw-rw-r--  1 ss ss 82005 Jan 10 07:03 composer.lock
-rw-rw-r--  1 ss ss   213 Jan 10 07:03 .editorconfig
drwxrwxr-x  8 ss ss  4096 Jan 10 07:03 .git
-rw-rw-r--  1 ss ss    14 Jan 10 07:03 .gitattributes
drwxrwxr-x  2 ss ss  4096 Jan 10 07:03 .github
```

```
-rw-rw-r--  1 ss ss   154 Jan 10 07:03 .gitignore
-rw-rw-r--  1 ss ss   681 Jan 10 07:07 Homestead.yaml
-rw-rw-r--  1 ss ss   265 Jan 10 07:03 init.bat
-rw-rw-r--  1 ss ss   250 Jan 10 07:03 init.sh
-rw-rw-r--  1 ss ss  1077 Jan 10 07:03 LICENSE.txt
-rw-rw-r--  1 ss ss   383 Jan 10 07:03 phpunit.xml.dist
-rw-rw-r--  1 ss ss  1404 Jan 10 07:03 readme.md
drwxrwxr-x  3 ss ss  4096 Jan 10 07:03 resources
drwxrwxr-x  2 ss ss  4096 Jan 10 07:03 scripts
drwxrwxr-x  4 ss ss  4096 Jan 10 07:03 src
drwxrwxr-x  4 ss ss  4096 Jan 10 07:03 tests
-rw-rw-r--  1 ss ss   277 Jan 10 07:03 .travis.yml
-rw-rw-r--  1 ss ss  1878 Jan 10 07:03 Vagrantfile
```

Configuring Homestead

Now you will configure Homestead using the `Homestead.yaml` file. If you are familiar with using terminal text editors such as Nano or Vim, you can go ahead and use your favorite tool. Or, you can use this command:

```
//code 2.24
ss@ss-H81M-S1:~/Homestead$ sudo gedit Homestead.yaml
```

This will open the `Homestead.yaml` file. The `provider` key in your `Homestead.yaml` file indicates which Vagrant provider should be used; it will be `virtualbox` or any other. You can set this to the provider you prefer as shown here:

```
//code 2.25
---
ip: "192.168.10.10"
memory: 2048
cpus: 1
provider: virtualbox

authorize: ~/.ssh/id_rsa.pub
```

```
keys:
    - ~/.ssh/id_rsa

folders:
    - map: ~/code
      to: /home/vagrant/code

sites:
    - map: homestead.test
      to: /home/vagrant/code/public

databases:
    - homestead

# ports:
#     - send: 50000
#       to: 5000
#     - send: 7777
#       to: 777
#       protocol: udp

# blackfire:
#     - id: foo
#       token: bar
#       client-id: foo
#       client-token: bar

# zray:
#  If you've already freely registered Z-Ray, you can place the token here.
#     - email: foo@bar.com
#       token: foo
#  Don't forget to ensure that you have 'zray: "true"' for your site.
```

Shared Folders and Homestead

In the previous section, I showed how to set `virtualbox` as the provider. I also set the shared folders that point to the Homestead environment. Take a look at these lines (from code 2.25):

```
folders:
    - map: ~/code
      to: /home/vagrant/code

sites:
    - map: homestead.test
      to: /home/vagrant/code/public

databases:
- homestead
```

The previous code says I should keep my code repositories in the `/home/ss/code` folder (`~/code`).

However, you can change this configuration, and the files in these folders will be kept in sync between your local machine and the Homestead environment. You may configure as many shared folders as necessary.

Finally, I have decided to start with two shared folders that will host two Laravel applications and assign two MySQL databases to them. So, these lines in my `Homestead.yaml` file look like this:

```
//code 2.26
// Homestead.yaml
---
ip: "192.168.10.10"
memory: 2048
cpus: 1
provider: virtualbox

authorize: ~/.ssh/id_rsa.pub

keys:
    - ~/.ssh/id_rsa

folders:
    - map: ~/code
      to: /home/vagrant/code

    - map: ~/code
      to: /home/vagrant/code
```

```
sites:
    - map: test.localhost
      to: /home/vagrant/code/blog/public

    - map: my.local
      to: /home/vagrant/code/larastartofinish/public

databases:
    - homestead
- myappo
```

In the previous code, you can see these repeated lines:

```
folders:
    - map: ~/code
      to: /home/vagrant/code

    - map: ~/code
      to: /home/vagrant/code
```

This is not just repetitive code; it means that for both projects I have chosen the `/home/vagrant/code` folder.

If you are following along, now you can run the Homestead environment and test your applications locally. You can type either `test.localhost` or `my.local URL` in your browser, and that will run your Laravel applications. However, before that, you need to accomplish one major task.

You must add these domains for your sites to the `hosts` file on your machine. The `hosts` file will redirect requests for your Homestead environment sites into your Homestead environment. On macOS/Linux, the `hosts` file is located in `/etc/hosts`.

So, type this command:

```
//code 2.27
//editing /etc/hosts file
$ sudo gedit /etc/hosts
```

This will give you the following output:

```
//output of code 2.27
127.0.0.1    localhost
```

```
::1     ip6-localhost ip6-loopback
127.0.1.1    ss-H81M-S1
127.0.0.1    sandbox.dev

# The following lines are desirable for IPv6 capable hosts
::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

On Windows, this is located in `C:\Windows\System32\drivers\etc\hosts`. The lines you add to this file will look like the following:

```
192.168.10.10      test.localhost
192.168.10.10      my.local
```

Please note that the IP addresses listed are the same as the ones in your `Homestead.yaml` file. Once you have added domains to your `hosts` file, you can launch the Vagrant box.

Launching the Vagrant Box

Launching the Vagrant box will enable you to access the site via your browser.

```
http://test.localhost
http://my.local
```

Launching the Vagrant box is easy. By staying in the `Homestead` folder, issue this command:

```
//code 2.28
$ vagrant up --provision
```

Why have I added `--provision` to the `vagrant up` command?

You need to understand one key concept regarding the `sites` property of Homestead. Originally, the `Homestead.yaml` file has these lines of code:

```
//code 2.2
- map: ~/code
      to: /home/vagrant/code
```

```
sites:
    - map: homestead.test
      to: /home/vagrant/code/public
```

But you have changed these lines to this:

```
//code 2.30
    - map: ~/code
      to: /home/vagrant/code

    - map: ~/code
      to: /home/vagrant/code

sites:
    - map: test.localhost
      to: /home/vagrant/code/blog/public

    - map: my.local
      to: /home/vagrant/code/larastartofinish/public
```

If you change the original `sites` property of Homestead, you need to provision the change to ensure it is applied.

You could have issued a command like `vagrant reload --provision` to update the Nginx configuration on the virtual machine, and then you can issue the `vagrant up` command. If it does not work, each time you need to add the `--provision` flag with your `vagrant up` command.

```
//code 2.31
ss@ss-H81M-S1:~/Homestead$ vagrant up --provision
Bringing machine 'homestead-7' up with 'virtualbox' provider...
==> homestead-7: Checking if box 'laravel/homestead' version '6.4.0' is up
                 to date...
==> homestead-7: Clearing any previously set forwarded ports...
==> homestead-7: Vagrant has detected a configuration issue which exposes a
==> homestead-7: vulnerability with the installed version of VirtualBox. The
==> homestead-7: current guest is configured to use an E1000 NIC type for a
==> homestead-7: network adapter which is vulnerable in this version of
                 VirtualBox.
```

```
==> homestead-7: Ensure the guest is trusted to use this configuration or
                 update
==> homestead-7: the NIC type using one of the methods below:
==> homestead-7:
==> homestead-7: https://www.vagrantup.com/docs/virtualbox/configuration.
                 html#default-nic-type
==> homestead-7: https://www.vagrantup.com/docs/virtualbox/networking.
                 html#virtualbox-nic-type
==> homestead-7: Clearing any previously set network interfaces...
==> homestead-7: Preparing network interfaces based on configuration...
    homestead-7: Adapter 1: nat
    homestead-7: Adapter 2: hostonly
==> homestead-7: Forwarding ports...
    homestead-7: 80 (guest) => 8000 (host) (adapter 1)
    homestead-7: 443 (guest) => 44300 (host) (adapter 1)
    homestead-7: 3306 (guest) => 33060 (host) (adapter 1)
    homestead-7: 4040 (guest) => 4040 (host) (adapter 1)
    homestead-7: 5432 (guest) => 54320 (host) (adapter 1)
    homestead-7: 8025 (guest) => 8025 (host) (adapter 1)
    homestead-7: 27017 (guest) => 27017 (host) (adapter 1)
    homestead-7: 22 (guest) => 2222 (host) (adapter 1)
==> homestead-7: Running 'pre-boot' VM customizations...
==> homestead-7: Booting VM...
==> homestead-7: Waiting for machine to boot. This may take a few minutes...
    homestead-7: SSH address: 127.0.0.1:2222
...
```

The process is not yet complete. It will fire up your Homestead development environment application, and finally, you need to issue this command:

```
//code 2.32
ss@ss-H81M-S1:~/Homestead$ vagrant ssh
Welcome to Ubuntu 18.04.1 LTS (GNU/Linux 4.15.0-38-generic x86_64)

* FYI Vagrant v2.2.2 & Virtualbox:
 * https://twitter.com/HomesteadDev/status/1071471881256079362
```

```
259 packages can be updated.
73 updates are security updates.
Last login: Sat Jan 12 05:18:55 2019 from 10.0.2.2
vagrant@homestead:~$ cd code/larastartofinish/
```

As shown in the previous code, you can now access your `/home/ss/code/larastartofinish` project through the `sites` property of Homestead. The `/vagrant/code` directory communicates with the `/home/ss/code/larastartofinish` project, where you have installed one Laravel application. I will discuss how to start the Laravel application in the next section. Before that, I would like to give you some tips about the database functionalities.

Homestead and MySQL

The Homestead development environment is actually a guest addition to your host machine. So, you should not try to communicate with your host MySQL database from the guest Homestead. Remember one key concept: Homestead has a lot of database support. MySQL is the default database. The username is `homestead`, and the password is `secret`. So, staying in the `vagrant` directory, `vagrant@homestead:~$`, you can just type this:

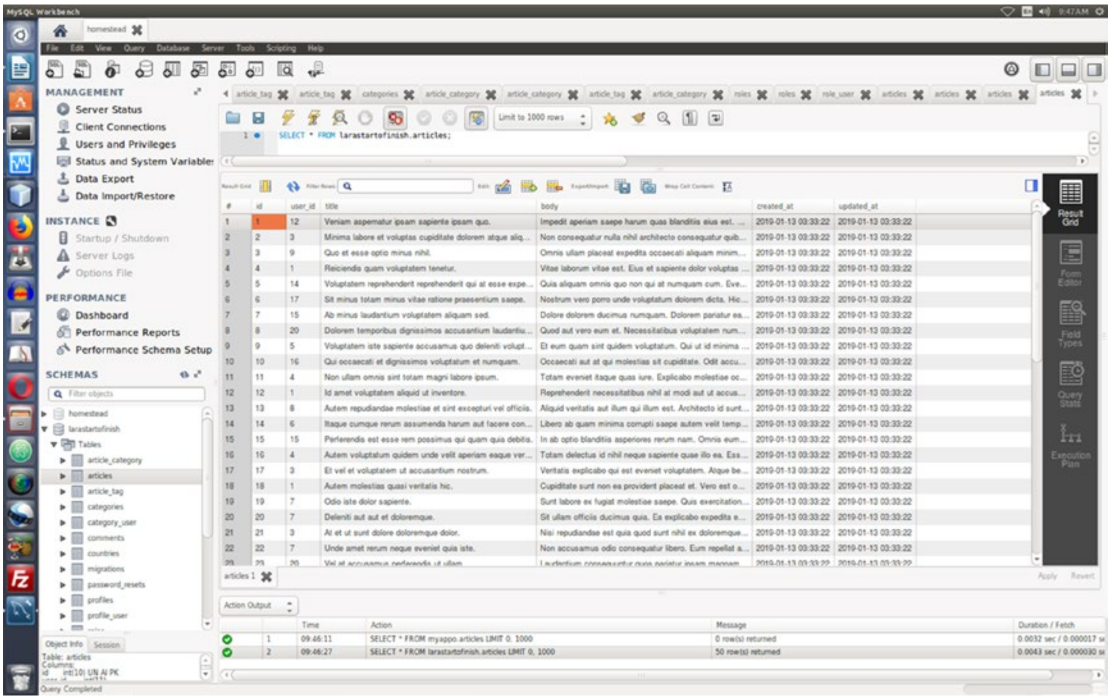
```
//code 2.33
vagrant@homestead:~$  mysql -u homestead -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 5
Server version: 5.7.24-0ubuntu0.18.04.1 (Ubuntu)

Copyright (c) 2000, 2018, Oracle and/or its affiliates. All rights
reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input
statement.
```

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| homestead          |
| larastartofinish   |
| myappo             |
| mysql              |
| performance_schema |
| socket_wrench      |
| sys                |
+--------------------+
8 rows in set (0.08 sec)

mysql>
```

You can see, I have already created two databases in my Homestead MySQL database in a driver. If you want to exit this terminal and shut down your Homestead development environment, you can issue this command:

```
//code 2.34
mysql> exit
Bye
vagrant@homestead:~$ exit
logout
Connection to 127.0.0.1 closed.
ss@ss-H81M-S1:~/Homestead$ vagrant halt
==> homestead-7: Attempting graceful shutdown of VM...
ss@ss-H81M-S1:~/Homestead$
```

So, you have successfully installed and closed down your Homestead development environment. If you want to access your MySQL database through a GUI, then consider installing MySQL Workbench, as shown in Figure 2-1.



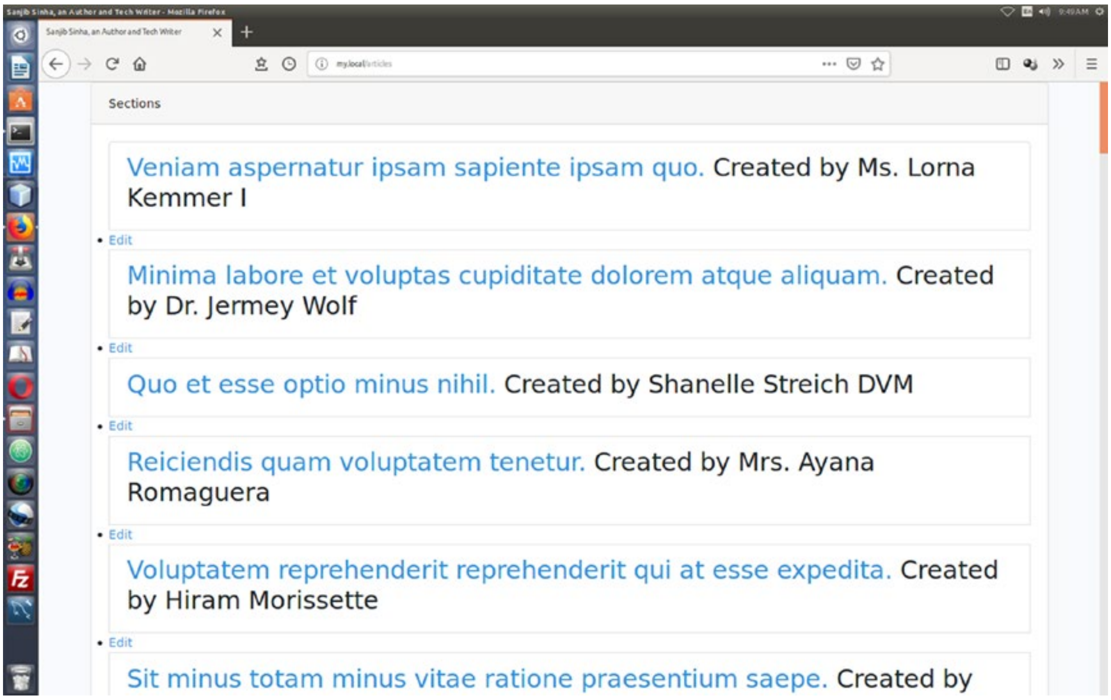
```
SELECT * FROM larastartofinish.articles;
```

#	id	user_id	title	body	created_at	updated_at
1	1	12	Veniam aspernatur ipsam sapiente ipsam quo.	Impedit aperiam saepe harum quas blanditiis eius est. ...	2019-01-13 03:33:22	2019-01-13 03:33:22
2	2	3	Minima labore et voluptas cupiditate dolorem atque aliq...	Non consequatur nulla nihil architecto consequatur quib...	2019-01-13 03:33:22	2019-01-13 03:33:22
3	3	9	Quo et esse optio minus nihil.	Omnis ullam placeat expedita occaecati aliquam minim...	2019-01-13 03:33:22	2019-01-13 03:33:22
4	4	1	Reiciendis quam voluptatem tenetur.	Vitae laborum vitae est. Eius et sapiente dolor voluptas ...	2019-01-13 03:33:22	2019-01-13 03:33:22
5	5	14	Voluptatem reprehenderit reprehenderit qui at esse expe...	Quia aliquam omnis quo non qui at numquam cum. Eve...	2019-01-13 03:33:22	2019-01-13 03:33:22
6	6	17	Sit minus totam minus vitae ratione praesentium saepe.	Nostrum vero porro unde voluptatum dolorem dicta. Hic...	2019-01-13 03:33:22	2019-01-13 03:33:22
7	7	15	Ab minus laudantium voluptatem aliquam sed.	Dolore dolorem ducimus numquam. Dolorem pariatur ea...	2019-01-13 03:33:22	2019-01-13 03:33:22
8	8	20	Dolorem temporibus dignissimos accusantium laudantiu...	Quod aut vero eum et. Necessitatibus voluptatem num...	2019-01-13 03:33:22	2019-01-13 03:33:22
9	9	5	Voluptatem iste sapiente accusamus quo deleniti volupt...	Et eum quam sint quidem voluptatum. Qui ut id minima ...	2019-01-13 03:33:22	2019-01-13 03:33:22
10	10	16	Qui occaecati et dignissimos voluptatum et numquam.	Occaecati aut at qui molestias sit cupiditate. Odit accu...	2019-01-13 03:33:22	2019-01-13 03:33:22
11	11	4	Non ullam omnis sint totam magni labore ipsum.	Totam eveniet itaque quas iure. Explicabo molestiae oc...	2019-01-13 03:33:22	2019-01-13 03:33:22
12	12	1	Id amet voluptatem aliquid ut inventore.	Reprehenderit necessitatibus nihil at modi aut ut accus...	2019-01-13 03:33:22	2019-01-13 03:33:22
13	13	8	Autem repudiandae molestiae et sint excepturi vel officiis.	Aliquid veritatis aut illum qui illum est. Architecto id sunt...	2019-01-13 03:33:22	2019-01-13 03:33:22
14	14	6	Itaque cumque rerum assumenda harum aut facere con...	Libero ab quam minima corrupti saepe autem velit temp...	2019-01-13 03:33:22	2019-01-13 03:33:22
15	15	15	Perferendis est esse rem possimus qui quam quia debitis.	In ab optio blanditiis asperiores rerum nam. Omnis eum...	2019-01-13 03:33:22	2019-01-13 03:33:22
16	16	4	Autem voluptatum quidem unde velit aperiam eaque ver...	Totam delectus id nihil neque sapiente quae illo ea. Ess...	2019-01-13 03:33:22	2019-01-13 03:33:22
17	17	3	Et vel et voluptatem ut accusantium nostrum.	Veritatis explicabo qui est eveniet voluptatem. Atque be...	2019-01-13 03:33:22	2019-01-13 03:33:22
18	18	1	Autem molestias quasi veritatis hic.	Cupiditate sunt non ea provident placeat et. Vero est o...	2019-01-13 03:33:22	2019-01-13 03:33:22
19	19	7	Odio iste dolor sapiente.	Sunt labore ex fugiat molestiae saepe. Quis exercitation...	2019-01-13 03:33:22	2019-01-13 03:33:22
20	20	7	Deleniti aut aut et doloremque.	Sit ullam officiis ducimus quia. Ea explicabo expedita e...	2019-01-13 03:33:22	2019-01-13 03:33:22
21	21	3	At et ut sunt dolore doloremque dolor.	Nisi repudiandae est quia quod sunt nihil ex doloremque...	2019-01-13 03:33:22	2019-01-13 03:33:22
22	22	7	Unde amet rerum neque eveniet quia iste.	Non accusamus odio consequatur libero. Eum repellat a...	2019-01-13 03:33:22	2019-01-13 03:33:22
23	23	20	Vel et accusamus perferendis ut ullam.	Laudantium consequuntur quos pariatur ipsam magnam	2019-01-13 03:33:22	2019-01-13 03:33:22

		Time	Action	Message	Duration / Fetch
✓	1	09:46:11	SELECT * FROM myappo.articles LIMIT 0, 1000	0 row(s) returned	0.0032 sec / 0.000017 se
✓	2	09:46:27	SELECT * FROM larastartofinish.articles LIMIT 0, 1000	50 row(s) returned	0.0043 sec / 0.000030 se

Figure 2-1. *MySQL Workbench*

I have already installed a Laravel application in my Homestead development environment and populated the database with some fake data. You can see the same `articles` table in a browser at `http://my.local/articles`, as shown in Figure 2-2.


Sections

Veniam aspernatur ipsam sapiente ipsam quo. Created by Ms. Lorna Kemmer I

- Edit

Minima labore et voluptas cupiditate dolorem atque aliquam. Created by Dr. Jermey Wolf

- Edit

Quo et esse optio minus nihil. Created by Shanelle Streich DVM

- Edit

Reiciendis quam voluptatem tenetur. Created by Mrs. Ayana Romaguera

- Edit

Voluptatem reprehenderit reprehenderit qui at esse expedita. Created by Hiram Morissette

- Edit

Sit minus totam minus vitae ratione praesentium saepe. Created by

Figure 2-2. *Your first Laravel project*

In the next section, I will discuss how to create your first Laravel project in your Homestead development environment.

Presuming you are about to learn Laravel from scratch, I will keep the next section brief. I will discuss how you can start your project. Later, in the coming chapters, you will learn how to build a database-driven application that will handle complex relations between various tables, creating, updating, and deleting records. It will be a company/project/task management system where users can also write articles or blog posts, write reviews about the companies, and do much more. There will be many roles, such as administrator, moderator, editor, and simple members or users who can register and log into the application.

How to Create a New Laravel Project

You'll name your application `larastartofinish`. This is an abbreviation of the full project name "Laravel Start to Finish." The `code` directory will be mapped to the Homestead development environment. So, from now on, you will install your Laravel applications there.

Staying in the `/home/ss/code` directory, you can install your Laravel application by issuing this command:

```
//code 2.35
$ composer create-project --prefer-dist laravel/laravel larastartofinish
```

This will install a fresh Laravel application in your `/home/ss/code` directory. Next, issue the following command for macOS/Linux in a terminal:

```
//code 2.36
$ sudo rm -rf vendor/ composer.lock
```

Basically, you have removed the Laravel dependencies that ship with Laravel, because you need to freshly install the new dependencies for your new `larastartofinish` application. So, issue this command:

```
//code 2.37
$ composer install
```

This will again install the necessary `vendor` folder and `composer.lock` file. You have successfully installed a new Laravel application in the `/home/ss/code` directory. Now you can start your Homestead development environment and start working on this application.

Staying in the Homestead folder, issue the following command:

```
vagrant up --provision
```

Next, issue the following command to start your Homestead development environment:

```
vagrant ssh
```

Then, change your directory like this so that it points to the `/home/ss/code/larastartofinish` directory:

```
//code 2.38
vagrant@homestead:~$ cd code/larastartofinish/
```

Next, you do not need to start your local server here. The advantage of Homestead is that now you can type `http://my.local` in your browser and view your new Laravel application. Next, you can create your first controller here by issuing this command:

```
//code 2.39
vagrant@homestead:~/code/larastartofinish$ php artisan make:controller
ArticleController --resource -model=Article
Controller created successfully.
```

This will create an `Article` controller, which is related to the `Article` model.

So, you can now successfully start working on your Homestead development environment using VirtualBox and Vagrant.

In the next chapter, you will start building this application from scratch and also learn how the route, controller, template, and view work.

CHAPTER 3

Routing, Controllers, Templates, and Views

In the previous two chapters, you learned how to create your environment so that you can use Composer to install fresh Laravel applications, and you learned many more other nitty-gritty details of Laravel. You also are now familiar with the concepts of the Model-View-Controller (MVC) logic system.

In this chapter, you will learn how Laravel follows the MVC pattern.

To enter an application, you need to have an entry point. The basic algebraic definition of *function* works here: you give input to a function, and you get output. When applying this to Laravel, you can replace the word *input* with *request* and replace the word *output* with *response*.

To start creating a Laravel application, the `web.php` file in the `routes` directory is important. It takes the requests from you, the user, and sends them to either a closure that gives a response, a view page that displays the response, or a controller that does the same thing; in some critical cases, the controller consults with the model and then is updated by the business logic.

You will learn all about this mechanism in this chapter.

Route Definitions

Laravel must know how to respond to a particular request, which relates to the concept of routing. Here is a basic example of routing a request:

```
//code 3.1
//routes/web.php
Route::get('/', function () {
    return view('welcome');
});
```

© Sanjib Sinha 2019
S. Sinha, *Beginning Laravel*, https://doi.org/10.1007/978-1-4842-4991-8_3

In this example, the Laravel route accepts a URI through the HTTP GET method using a closure. The closure (also called an *anonymous function*) returns a view page. I will come back to that in a minute, but I need to point out one more thing here: the `Route` class uses a static method, which is a class-level method. Why does it not use an instance of `Route`? This is because a class-level variable consumes less memory than objects, whereas an object, once instantiated, starts consuming memory. This is an important concept of object-oriented programming.

This file defines the application's URL endpoints so Laravel knows how to respond to that request. Here the URI is the document root or home page, and the Closure or anonymous function returns a view that contains an HTML page. These views are called *view pages*, and they are based on the Blade template engine that Laravel ships with.

This is a simple and expressive method of defining routes where models and controllers are absent. However, in a real-world scenario, you keep controllers between models and views, and the routes are initiated by the controllers. You will see an example of that in a few minutes.

You could have written the previous code in this way:

```
//code 3.2
//routes/web.php
Route::get('/', 'WelcomeController@index');
```

In the previous code, inside the welcome controller's `index` method, you can return the same view.

How to Find the Default Route Files

The Laravel framework automatically loads the route files in the `routes` directory. For the web interface, the `routes/web.php` file defines the routes. In Chapter 3, you will see how these routes are assigned to the `web` middleware group. Actually, route provides many features such as session state and CSRF protection. There is another middleware group, called `api`. I will also discuss this group later in this chapter.

In most cases, you will start by using this `routes/web.php` file. However, using closures is not a workable option. In some special cases, you will definitely use closures, but you will normally use a dedicated controller to manage a connected resource, as I showed in the previous code snippet.

Let's think about a `TaskController`. This controller might retrieve all `tasks`. It gets all `tasks` views, inserts a new task, edits and updates an existing task, and finally deletes an existing list from the database.

Route and RESTful Controller

Since you will normally use a dedicated controller to manage a connected resource, the concept of RESTful or resourceful controllers applies here. In the "Resourceful Controller" section, I will discuss resourceful controllers in more detail, but before that, all you need to know is that a RESTful or resourceful controller can handle all seven routes associated with it. For now, you should know that when you use a create-read-update-delete (CRUD) approach, you use HTTP methods such as GET, POST, PUT, and DELETE. How do you get seven routes from this? Well, you use GET four times: to show all content, to show a form to create new content, to show a particular piece of content (ideally getting that by its ID), and finally to show the edit form to update the content.

You can write a route like this:

```
//code 3.3
//routes/web.php
Route::resource('tasks', 'TaskController');
```

Now, you can view all the tasks by typing `http://example.com/tasks` in your browser because the controller is waiting for the request and is programmed to display all the tasks.

What does the term RESTful mean? There is a fundamental difference between a web application and a REST API. When you get the response from a web application, it generally consists of HTML, CSS, and JavaScript. On the other hand, the REST API returns data in the form of JSON or XML. As you progress, you will learn more about the close relationship between Laravel and JSON output.

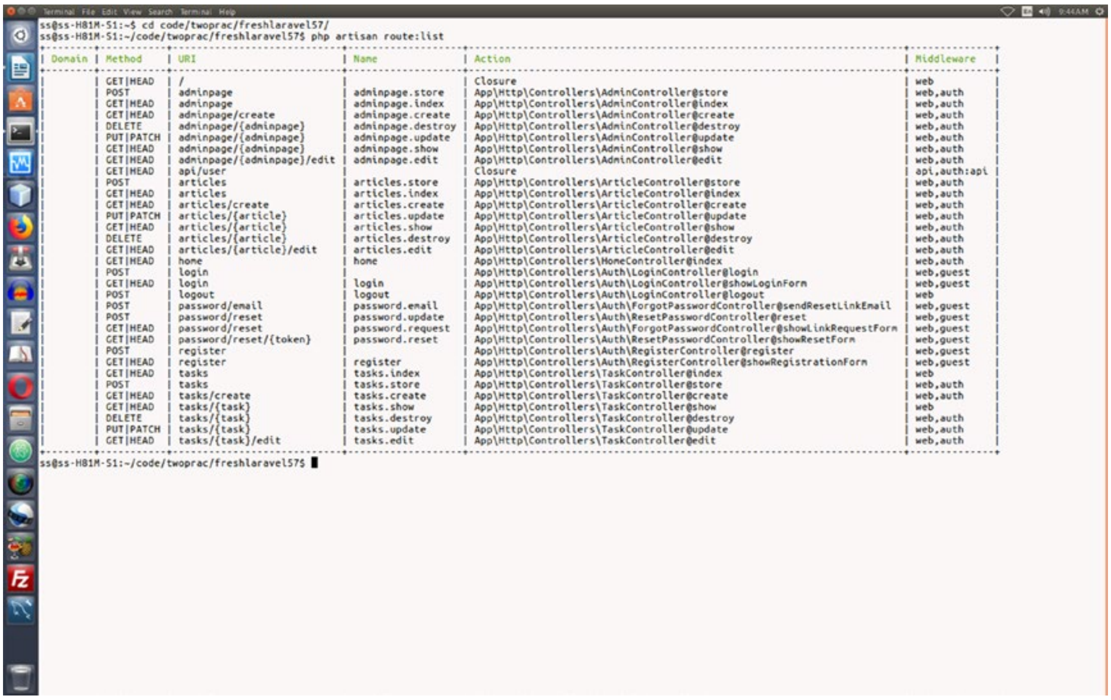
How to List All Routes

In a large application, it is quite cumbersome to maintain a list of all the routes. There might be hundreds of routes, and they are connected to dedicated resources with separate business logic.

Laravel has made it easy to list all routes by using a single command. I will show you one small application where I have maintained a few controllers and view pages. There is an administrator section, and I can create, retrieve, update, and delete (CRUD) articles through that admin panel, and I can also add some tasks. For doing these simple operations, I have created three controllers and made them all resourceful. Now if you issue a single command to get the route lists, like this:

```
//code 3.4
$ php artisan route:list
```

it will give you some output like Figure 3-1.



```
ss@ss-H81M-S1:~$ cd code/twoprac/freshlaravel57/
ss@ss-H81M-S1:~/code/twoprac/freshlaravel57$ php artisan route:list
+--------+-----------+----------------------------+-------------------+------------------------------------------------------------------------+--------------+
| Domain | Method    | URI                        | Name              | Action                                                                 | Middleware   |
+--------+-----------+----------------------------+-------------------+------------------------------------------------------------------------+--------------+
|        | GET|HEAD  | /                          |                   | Closure                                                                | web          |
|        | POST      | adminpage                  | adminpage.store   | App\Http\Controllers\AdminController@store                             | web,auth     |
|        | GET|HEAD  | adminpage                  | adminpage.index   | App\Http\Controllers\AdminController@index                             | web,auth     |
|        | GET|HEAD  | adminpage/create           | adminpage.create  | App\Http\Controllers\AdminController@create                            | web,auth     |
|        | DELETE    | adminpage/{adminpage}      | adminpage.destroy | App\Http\Controllers\AdminController@destroy                           | web,auth     |
|        | PUT|PATCH | adminpage/{adminpage}      | adminpage.update  | App\Http\Controllers\AdminController@update                            | web,auth     |
|        | GET|HEAD  | adminpage/{adminpage}      | adminpage.show    | App\Http\Controllers\AdminController@show                              | web,auth     |
|        | GET|HEAD  | adminpage/{adminpage}/edit | adminpage.edit    | App\Http\Controllers\AdminController@edit                              | web,auth     |
|        | GET|HEAD  | api/user                   |                   | Closure                                                                | api,auth:api |
|        | POST      | articles                   | articles.store    | App\Http\Controllers\ArticleController@store                           | web,auth     |
|        | GET|HEAD  | articles                   | articles.index    | App\Http\Controllers\ArticleController@index                           | web,auth     |
|        | GET|HEAD  | articles/create            | articles.create   | App\Http\Controllers\ArticleController@create                          | web,auth     |
|        | PUT|PATCH | articles/{article}         | articles.update   | App\Http\Controllers\ArticleController@update                          | web,auth     |
|        | GET|HEAD  | articles/{article}         | articles.show     | App\Http\Controllers\ArticleController@show                            | web,auth     |
|        | DELETE    | articles/{article}         | articles.destroy  | App\Http\Controllers\ArticleController@destroy                         | web,auth     |
|        | GET|HEAD  | articles/{article}/edit    | articles.edit     | App\Http\Controllers\ArticleController@edit                            | web,auth     |
|        | GET|HEAD  | home                       | home              | App\Http\Controllers\HomeController@index                              | web,auth     |
|        | POST      | login                      |                   | App\Http\Controllers\Auth\LoginController@login                        | web,guest    |
|        | GET|HEAD  | login                      | login             | App\Http\Controllers\Auth\LoginController@showLoginForm                | web,guest    |
|        | POST      | logout                     | logout            | App\Http\Controllers\Auth\LoginController@logout                       | web          |
|        | POST      | password/email             | password.email    | App\Http\Controllers\Auth\ForgotPasswordController@sendResetLinkEmail  | web,guest    |
|        | POST      | password/reset             | password.update   | App\Http\Controllers\Auth\ResetPasswordController@reset                | web,guest    |
|        | GET|HEAD  | password/reset             | password.request  | App\Http\Controllers\Auth\ForgotPasswordController@showLinkRequestForm | web,guest    |
|        | GET|HEAD  | password/reset/{token}     | password.reset    | App\Http\Controllers\Auth\ResetPasswordController@showResetForm        | web,guest    |
|        | POST      | register                   |                   | App\Http\Controllers\Auth\RegisterController@register                  | web,guest    |
|        | GET|HEAD  | register                   | register          | App\Http\Controllers\Auth\RegisterController@showRegistrationForm      | web,guest    |
|        | GET|HEAD  | tasks                      | tasks.index       | App\Http\Controllers\TaskController@index                              | web          |
|        | POST      | tasks                      | tasks.store       | App\Http\Controllers\TaskController@store                              | web,auth     |
|        | GET|HEAD  | tasks/create               | tasks.create      | App\Http\Controllers\TaskController@create                             | web,auth     |
|        | GET|HEAD  | tasks/{task}               | tasks.show        | App\Http\Controllers\TaskController@show                               | web          |
|        | DELETE    | tasks/{task}               | tasks.destroy     | App\Http\Controllers\TaskController@destroy                            | web,auth     |
|        | PUT|PATCH | tasks/{task}               | tasks.update      | App\Http\Controllers\TaskController@update                             | web,auth     |
|        | GET|HEAD  | tasks/{task}/edit          | tasks.edit        | App\Http\Controllers\TaskController@edit                               | web,auth     |
+--------+-----------+----------------------------+-------------------+------------------------------------------------------------------------+--------------+
ss@ss-H81M-S1:~/code/twoprac/freshlaravel57$
```

***Figure 3-1.** List of all routes in an application*

Figure 3-1 shows a table where the column names are Method, URI, Name, Action, and Middleware. In the following code, you will see the method name first, which is either GET, POST, PUT, or DELETE. After that, you will see the URI, such as `/adminpage`; next comes the view page names like `adminpage.show`. This is followed by the action or controller methods, like `AdminController@index`, and at the end comes the middleware. I will talk about the middleware later in the book in great detail.

```
ss@ss-H81M-S1:~/code/twoprac/freshlaravel57$ php artisan route:list
+--------+-----------+-----------------------------+-------------------+
---------------------------------------------------------------------+
--------------+
| Domain | Method    | URI                         | Name              |
Action
| Middleware   |
+--------+-----------+-----------------------------+-------------------+
---------------------------------------------------------------------+
--------------+
|        | GET|HEAD  | /                           |                   |
Closure
| web          |
|        | GET|HEAD  | adminpage                   | adminpage.index   |
App\Http\Controllers\AdminController@index
| web,auth     |
|        | POST      | adminpage                   | adminpage.store   |
App\Http\Controllers\AdminController@store
| web,auth     |
|        | GET|HEAD  | adminpage/create            | adminpage.create  |
App\Http\Controllers\AdminController@create
| web,auth     |
|        | DELETE    | adminpage/{adminpage}       | adminpage.destroy |
App\Http\Controllers\AdminController@destroy
| web,auth     |
|        | PUT|PATCH | adminpage/{adminpage}       | adminpage.update  |
App\Http\Controllers\AdminController@update
| web,auth     |
|        | GET|HEAD  | adminpage/{adminpage}       | adminpage.show    |
App\Http\Controllers\AdminController@show
| web,auth     |
|        | GET|HEAD  | adminpage/{adminpage}/edit | adminpage.edit    |
App\Http\Controllers\AdminController@edit
| web,auth     |
|        | GET|HEAD  | api/user                   |                    |
Closure
| api,auth:api |
|        | POST      | articles                   | articles.store     |
App\Http\Controllers\ArticleController@store
| web,auth     |
|        | GET|HEAD  | articles                   | articles.index     |
App\Http\Controllers\ArticleController@index
| web,auth     |
|        | GET|HEAD  | articles/create            | articles.create    |
App\Http\Controllers\ArticleController@create
| web,auth     |
|        | PUT|PATCH | articles/{article}         | articles.update    |
App\Http\Controllers\ArticleController@update
| web,auth     |
|        | GET|HEAD  | articles/{article}         | articles.show      |
App\Http\Controllers\ArticleController@show
| web,auth     |
|        | DELETE    | articles/{article}         | articles.destroy   |
App\Http\Controllers\ArticleController@destroy
| web,auth     |
|        | GET|HEAD  | articles/{article}/edit    | articles.edit      |
App\Http\Controllers\ArticleController@edit
| web,auth     |
|        | GET|HEAD  | home                       | home               |
App\Http\Controllers\HomeController@index
| web,auth     |
|        | GET|HEAD  | login                      | login              |
App\Http\Controllers\Auth\LoginController@showLoginForm
| web,guest    |
|        | POST      | login                      |                    |
App\Http\Controllers\Auth\LoginController@login
| web,guest    |
|        | POST      | logout                     | logout             |
App\Http\Controllers\Auth\LoginController@logout
| web          |
|        | POST      | password/email              | password.email    |
App\Http\Controllers\Auth\ForgotPasswordController@sendResetLinkEmail
| web,guest    |
|        | GET|HEAD  | password/reset              | password.request  |
App\Http\Controllers\Auth\ForgotPasswordController@showLinkRequestForm
| web,guest    |
|        | POST      | password/reset              | password.update   |
App\Http\Controllers\Auth\ResetPasswordController@reset
| web,guest    |
|        | GET|HEAD  | password/reset/{token}      | password.reset    |
App\Http\Controllers\Auth\ResetPasswordController@showResetForm
| web,guest    |
|        | GET|HEAD  | register                    | register          |
App\Http\Controllers\Auth\RegisterController@showRegistrationForm
| web,guest    |
|        | POST      | register                    |                   |
App\Http\Controllers\Auth\RegisterController@register
| web,guest    |
|        | POST      | tasks                       | tasks.store       |
App\Http\Controllers\TaskController@store
| web,auth     |
|        | GET|HEAD  | tasks                       | tasks.index       |
App\Http\Controllers\TaskController@index
| web          |
|        | GET|HEAD  | tasks/create                | tasks.create      |
App\Http\Controllers\TaskController@create
| web,auth     |
|        | GET|HEAD  | tasks/{task}                | tasks.show        |
App\Http\Controllers\TaskController@show
| web          |
|        | DELETE    | tasks/{task}                | tasks.destroy     |
App\Http\Controllers\TaskController@destroy
| web,auth     |
|        | PUT|PATCH | tasks/{task}                | tasks.update      |
App\Http\Controllers\TaskController@update
| web,auth     |
|        | GET|HEAD  | tasks/{task}/edit          | tasks.edit          |
App\Http\Controllers\TaskController@edit
| web,auth     |
|        | GET|HEAD  | user                       |                     |
Closure
```

Creating Controllers, Views, and Managing Routes

I have already pointed out why you need a controller. You cannot define all of your request handling logic as closures in route files. Instead, you can organize this action using controller classes. Controller class group related request handling logic into a single class. You can create a controller quite easily by keeping the connected resources in mind. Controllers can be stored in the `app/Http/Controllers` directory.

Let's create a controller first, as shown here:

```
//code 3.5
$ php artisan make:controller TaskController --resource
//the output of code 3.5
Controller created successfully.
```

This is what the `app/Http/Controllers/TaskController.php` file looks like:

```
//code 3.6
// app/Http/Controllers/TaskController.php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class TaskController extends Controller
{
    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
```

```
public function index()
{
    //
}

/**
 * Show the form for creating a new resource.
 *
 * @return \Illuminate\Http\Response
 */
public function create()
{
    //
}

/**
 * Store a newly created resource in storage.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return \Illuminate\Http\Response
 */
public function store(Request $request)
{
    //
}

/**
 * Display the specified resource.
 *
 * @param  int  $id
 * @return \Illuminate\Http\Response
 */
public function show($id)
{
    //
}
```

```
    /**
     * Show the form for editing the specified resource.
     *
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function edit($id)
    {
        //
    }

    /**
     * Update the specified resource in storage.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function update(Request $request, $id)
    {
        //
    }

    /**
     * Remove the specified resource from storage.
     *
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function destroy($id)
    {
        //
    }
}
```

There are seven methods that you need to create a CRUD system for your connected resources. Here the connected resources are the `Task model`, through which you will handle your database and business logic. The other parts of resources are your related views page through which you will handle all types of front-end operations such as showing tasks and creating, editing, and deleting your tasks.

While creating the `TaskController`, I have added an extra parameter called `--resource`. Because of that parameter, I get all seven CRUD methods in my `TaskController`. I could have created them manually, but Laravel takes care of creating them automatically because I passed that parameter while creating the `Controller` class.

Now, in your `routes/web.php` file, you can address all associated routes in a single line of code, as shown here:

```
//code 3.7
//routes/web.php
Route::resource('tasks', 'TaskController');
```

CRUD and the Seven Methods

The single line of code shown previously handles all seven methods created by default in the `TaskController` class.

If you issue the `php artisan route:list` command, you will see the related URIs, names, actions, and methods.

Let's check it out. All seven methods, URIs, and names (related view pages) are as follows:

```
| GET|HEAD  | tasks                        | tasks.index
| POST      | tasks                        | tasks.store         |
| GET|HEAD  | tasks/create                 | tasks.create        |
| GET|HEAD  | tasks/{task}                 | tasks.show          |
| DELETE    | tasks/{task}                 | tasks.destroy       |
| PUT|PATCH | tasks/{task}                 | tasks.update        |
| GET|HEAD  | tasks/{task}/edit
```

Let's try to understand how it works. When you type a URI like `http://example.com/tasks` in your browser, The 'task resource' sends the GET requests. The `index.blade.php` page belonging to the `resources/views/tasks` folder displays all tasks.

The third method is a GET, and the URI is `http://example.com/tasks/create`. It will display a form in the `create.blade.php` page belonging to the `resources/views/tasks` folder. Here you will fill up all the related fields and hit the Submit button. Once you do that, the second method will start acting. That is the POST, and you do not need to have the `store.blade.php` file. Laravel handles this POST request automatically in the `TaskController` class.

And it goes on like this.

To get a complete view, you need to see what your final `TaskController.php` file looks like, as shown here:

```
//code 3.8
// app/Http/Controllers/TaskController.php
<?php

namespace App\Http\Controllers;

use App\Task;
use Illuminate\Support\Facades\Auth;
use Illuminate\Http\Request;

class TaskController extends Controller
{

    /**
   * Create a new controller instance.
   *
   * @return void
   */
  public function __construct()
  {
      $this->middleware('auth')->except('index', 'show');
  }
    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
```

```
public function index()
{
    //
    $tasks = Task::get();
    return view('tasks.index', compact('tasks'));
}

/**
 * Show the form for creating a new resource.
 *
 * @return \Illuminate\Http\Response
 */
public function create()
{
    //
    if(Auth::user()->is_admin == 1){
        return view('tasks.create');
    }
    else {
      return redirect('home');
    }

}

/**
 * Store a newly created resource in storage.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return \Illuminate\Http\Response
 */
public function store(Request $request)
{
    //
    if(Auth::user()->is_admin == 1){
          $post = new Task;
$post->title = $request->input('title');
$post->body = $request->input('body');
$post->save();

if($post){
           return redirect('tasks');
       }

      }

  }

  /**
   * Display the specified resource.
   *
   * @param  int  $id
   * @return \Illuminate\Http\Response
   */
  public function show($id)
  {
      //
      $task = Task::findOrFail($id);
      return view('tasks.show', compact('task'));
  }

  /**
   * Show the form for editing the specified resource.
   *
   * @param  int  $id
   * @return \Illuminate\Http\Response
   */
  public function edit($id)
  {
      //
       if(Auth::user()->is_admin == 1){
      $task = Task::findOrFail($id);
      return view('tasks.edit', compact('task'));
```

```
    }
    else {
      // code...
      return redirect('home');
    }
  }

  /**
   * Update the specified resource in storage.
   *
   * @param  \Illuminate\Http\Request  $request
   * @param  int  $id
   * @return \Illuminate\Http\Response
   */
  public function update(Request $request, $id)
  {
      //
      if(Auth::user()->is_admin == 1){

          $post = Task::findOrFail($id);
$post->title = $request->input('title');
$post->body = $request->input('body');
$post->save();

if($post){
           return redirect('tasks');
       }

      }

  }

  /**
   * Remove the specified resource from storage.
   *
   * @param  int  $id
   * @return \Illuminate\Http\Response
   */
```

```
    public function destroy($id)
    {
        //
    }
}
```

In the previous code, I skipped the `destroy` method, but it should give you an idea of how you can associate your controller class to a connected resource.

At the same time, you need to see the `Task` model and the database table that you have created so that this connection between the Model-Controller-View is complete.

First, while creating a model, you pass an extra parameter, `-m`, so that the database migration takes place automatically. Laravel creates the primary task and database table. (Of course, you can create them separately too.) While creating a connected resource with a controller class, it is a good idea to create the table with the `Task` model.

```
//code 3.9
//creating Task model and database table
$ php artisan make:model Task -m
Model created successfully.
Created Migration: 2019_02_16_041652_create_tasks_table
```

Now, you have successfully created the model and created the migration.

For brevity, I have kept this model and the migration tasks table quite simple. So, to create a model and related database table, you need to issue the command `php artisan make:model Task -m`. Only after that are the model and the related database table created.

Here is what the `Task` model looks like:

```
//code 3.10
//app/Task.php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Task extends Model
{
    //
    protected $fillable = [
        'title', 'body'
    ];
}
```

After running the previous command, you get a task table in the `database/migration` folder. Now you can add more functionality to that table by adding new columns. After the modification, you will issue another command so that Laravel knows that it should take the necessary steps to modify it.

You can modify the database migration tasks table as follows:

```
//code 3.11
// database/migrations/tasks table
<?php

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

class CreateTasksTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('tasks', function (Blueprint $table) {
            $table->increments('id');
            $table->string('title');
            $table->text('body');
            $table->timestamps();
        });
    }

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

Now, your route, model, controller, and view circle is complete. You will learn how each of these components works in later chapters. Understand that you should have the resources as separate as possible. They must be loosely coupled so that each controller has a connected model and view pages.

Models Acts as a Resource

Now through the `Model` class you can build bridges to the other parts of your application. But that is a different thing. I will discuss later how Eloquent relationships work and how each database table communicates with each other.

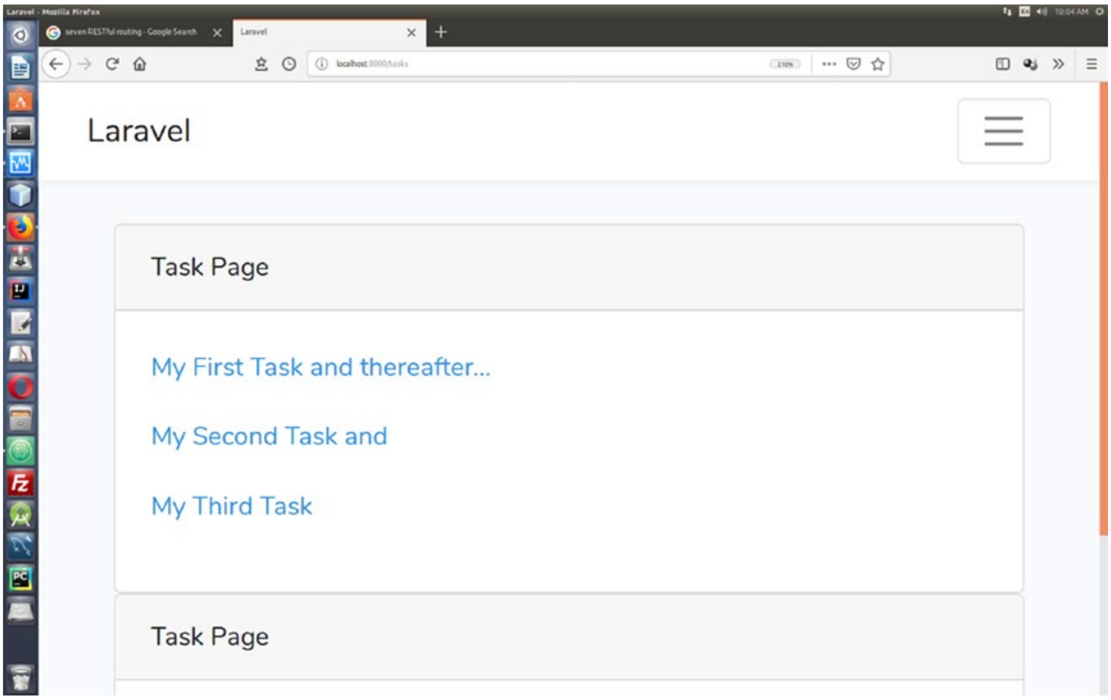
In the code for the `TaskController` class (code 3.8), you saw lines like this:

```
//code 3.12
public function __construct()
  {
      $this->middleware('auth')->except('index', 'show');
  }
```

The `TaskController` class opens up its two methods (`index` and `show`) only for guests or the public. To operate other methods, you need to be an authenticated user, and then you can also apply authorization and all the other related middleware actions.

In the "Authentication, Authorization, and Middleware" Chapter 8, I will discuss the methodologies behind the `TaskController` class. Until then, know that through your `Controller` class you can control your entire application. The greatness of Laravel is that it also gives you ample chances to control your application through other parts, such as your route file, view page, and the model.

Now, if you type `http://localhost:8000/tasks` in your browser, you will see the page in Figure 3-2.


Laravel

Task Page

My First Task and thereafter...

My Second Task and

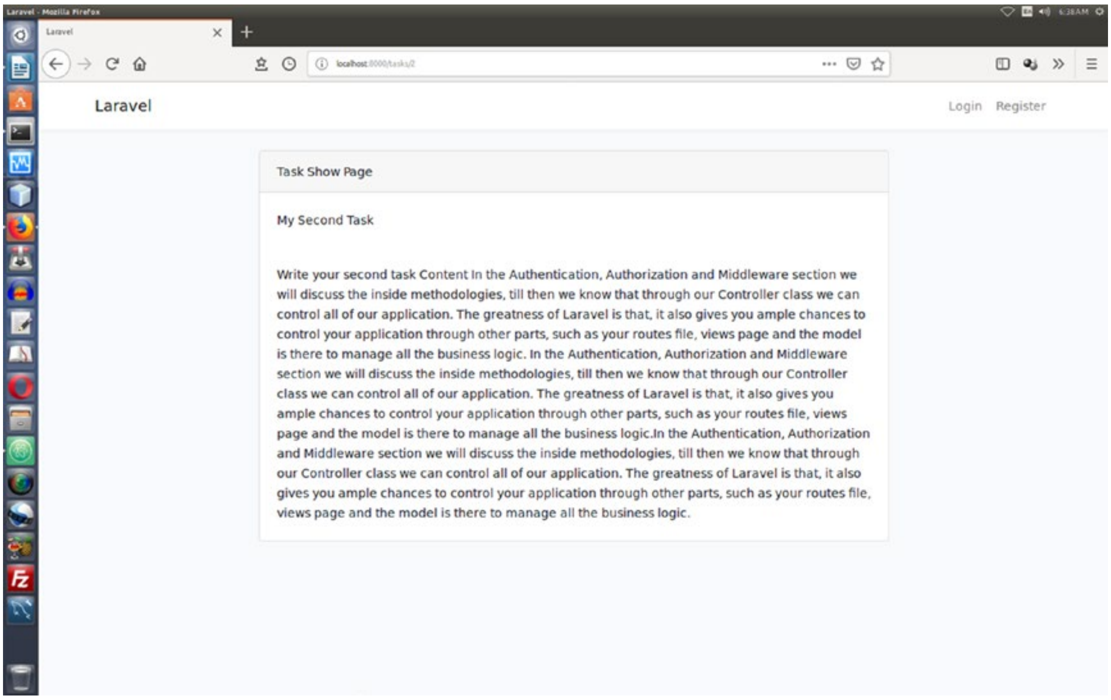
My Third Task

Task Page

Figure 3-2. *Showing all tasks*

In the next section, you will see how resourceful controllers come to your rescue while you try to maintain a separation of concerns.

And if you click the second task, you will see the view page shown in Figure 3-3.



Task Show Page

My Second Task

Write your second task Content In the Authentication, Authorization and Middleware section we will discuss the inside methodologies, till then we know that through our Controller class we can control all of our application. The greatness of Laravel is that, it also gives you ample chances to control your application through other parts, such as your routes file, views page and the model is there to manage all the business logic. In the Authentication, Authorization and Middleware section we will discuss the inside methodologies, till then we know that through our Controller class we can control all of our application. The greatness of Laravel is that, it also gives you ample chances to control your application through other parts, such as your routes file, views page and the model is there to manage all the business logic.In the Authentication, Authorization and Middleware section we will discuss the inside methodologies, till then we know that through our Controller class we can control all of our application. The greatness of Laravel is that, it also gives you ample chances to control your application through other parts, such as your routes file, views page and the model is there to manage all the business logic.

Figure 3-3. *The third task*

Now go to `http://localhost:8000/tasks/3`, which is the third task. In the `TaskController` class, the method is `show`.

Remember this part of the file `TaskController.php`:

```
//code 3.13
public function show($id)
    {
        //
        $task = Task::findOrFail($id);
        return view('tasks.show', compact('task'));
    }
```

This actually returns `show.blade.php` belonging to the `resources/views/tasks` folder, and the code snippet I have used there looks like this:

```
//code 3.14
// resources/views/tasks/show.blade.php
<div class="card-body">
```

```
            {{ $task->title }}
        </div>
        <div class="card-body">
            {{ $task->body }}
        </div>
```

I will discuss this part in a moment, so please keep reading.

Models Act As Resources

I would like to mention one thing: the `TaskController` class manages the `task` resource in such a way that you can view the `index.blade.php` and `show.blade.php` pages without being authenticated. However, you can do that in other parts of the same resource.

Suppose you want to type this URI: `http://localhost:8000/tasks/create`. This will take you to the login page. See Figure 3-4.

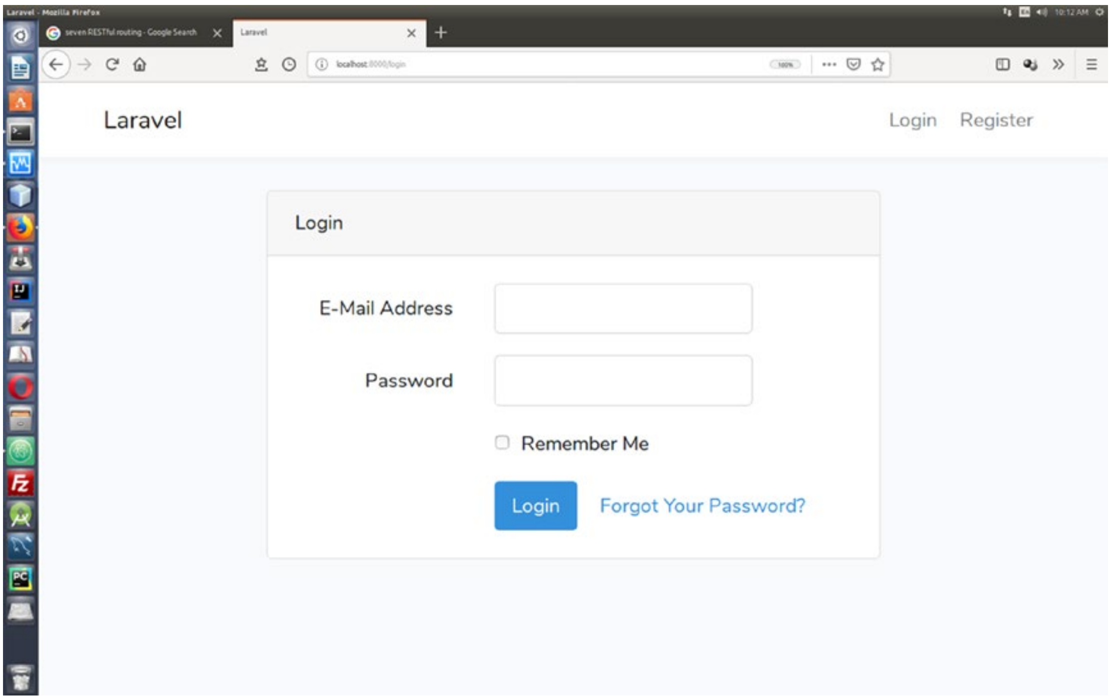


Figure 3-4. *The login page*

Why does this happen?

Remember this part of `TaskController.php`:

```
//code 3.15
public function create()
    {
        //
        if(Auth::user()->is_admin == 1){
            return view('tasks.create');
        }
        else {
          return redirect('home');
        }

    }
```

This clearly states that if the user is not `admin`, they cannot view this page. Otherwise, it redirects the user to the home page, which again redirects back to the login page.

I hope that you can now follow the logic behind Laravel's routes, controller, model, and view mechanism. In the route lists, you can see how these components are related to each other. For each request, whether that be GET or POST, Laravel handles it nicely and relates it to the respective URI. Then that URI fires up the respective RESTful `Controller` method, and that takes you to the respective view page.

In the next section, you will see how the RESTful or resourceful controller encapsulates those URI requests in a single line.

Resourceful Controllers

With a single line of code, how can you handle a typical CRUD route to a controller? Laravel's resource routing is the answer. You want to store every task in your database. You want to edit and update every task. You want to create a controller that handles all HTTP requests for tasks stored by your application.

You have already created the resource `TaskController` that way, and you have seen how it contains a method for each of the available resource operations. Next, you have registered a route to that controller, as shown here:

```
    Route::resource('tasks', 'TaskController');
```

Now, this single route declaration creates multiple routes to handle a variety of actions on the resource. If you open the `app/HTTP/Controllers/TaskController.php` file, you will find that each of these actions has related notes informing you of the HTTP verbs and URIs they handle.

Let's see one example of the `index` method:

```
//code 3.16
// app/HTTP/Controllers/TaskController.php
/**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        $tasks = Task::get();
        return view('tasks.index', compact('tasks'));
    }
```

This clearly states "Display a listing of the resource." At the same time, if you watch the route listing (shown next), you will find for the action `App\Http\Controllers\TaskController@index` that the name is `tasks.index` (this means the `index.blade.php` file belonging to the `tasks` folder). The method is GET, and the URI is `tasks`. This means if you type the URI `http://localhost:8000/tasks`, you will view all the tasks.

You can write all actions handled by the resource `TaskController` in one place in this way:

```
//code 3.17
Method    | URI                    | Action       | Route Name
----------|------------------------|--------------|---------------------
GET       | 'tasks'                | index        | tasks.index
GET       | 'tasks/create'         | create       | tasks.create
POST      | 'tasks'                | store        | tasks.store
GET       | 'tasks/{task}'         | show         | tasks.show
GET       | 'tasks/{task}/edit'    | edit         | tasks.edit
PUT/PATCH | 'tasks/{task}'         | update       | tasks.update
DELETE    | 'tasks/{task}'         | destroy      | tasks.destroy
```

The Importance of the Resourceful Controller

Creating controllers in Laravel is simple. In an MVC pattern, controllers play a vital role, and defining all your request handling logic as an anonymous function in a route file is not a viable solution for a big application. So, you will always use controllers as a transport medium. The main advantage of a controller is that it can group all the request handling into a single class, and that class can be stored in the `app/HTTP/Controllers` directory.

Another advanced feature is that you can register many resource controllers at once by passing an array to the `resources` method. It looks like this:

```
Route::resources([
    'myfiles' => 'MyController',
    'moreactions' => 'MoreController'
]);
```

A resourceful controller uses many types of HTTP verbs at the same time, such as GET, POST, PUT/PATCH, and DELETE. By using the GET verb, you can access four action methods in the controller; they are `index`, `create`, `show`, and `edit`. The action methods in the controller handle the route names of the same name; for example, the `index action` would handle a URI like `/myfiles` and a route name of `myfiles.index`. The `create` and `edit` actions only show web forms, so the GET verb is perfect for them.

You need a POST verb or method for the `store` action because you are sending data to the app, and behind the scenes, the Laravel service container (I will discuss this later in detail) takes the trouble to go through the model to insert new data in the related database table. The URI remains the same as `index`, like `/myfiles`, but the action method changes to `store`, and the route name also changes to `myfiles.store`. The HTTP verb PUT/PATCH is used for updating only a single record. Therefore, the URI changes to `/myfiles/{myfile}`, the action method changes to `update`, and the `Route` name changes to `myfiles.update`.

How to Supplement the Resource Controller

Sometimes you may want to add routes to a resource controller. A resource controller always handles a default set of resource routes; however, you may want a few more like this:

```
//code 3.18
Route::get('tasks/important', 'TaskController@important');
```

Of course, your resource controller will never create an extra method like this. You need to add it manually. However, you need to register this route before the resource route. Otherwise, the resource route will take precedence. In the `routes/web.php` file, you should write these lines of code:

```
//code 3.19
Route::get('tasks/important', 'TaskController@important');
Route::resource('tasks', 'TaskController');
```

However, you should keep your controller focused, and to do that, my suggestion is that it is better to maintain several small controllers so you don't have to juggle with the position in your route file.

Getting User Input and Dependency Injection

The Laravel service container is used to resolve all Laravel controllers. This means that when you create a resource controller, method injection normally occurs. How does this work?

It's simple. Just take a look at these lines of code in your `TaskController store` method:

```
//code 3.20
    public function store(Request $request)
    {
        if(Auth::user()->is_admin == 1){
          $post = new Task;          $post->title = $request-
                                     >input('title');
          $post->body = $request->input('body');
          $post->save();

          if($post){
             return redirect('tasks');
          }
        }
    }
```

The `$request` object from `\Illuminate\Http\Request` has automatically been injected and resolved by Laravel's service container. Behind the scenes, a form is sending data, and you can get the following code as a result:

```
$post->title = $request→input('title');
```

When you are going to edit any `task` item, the same thing takes place in the `update` method.

```
//code 3.21
    public function update(Request $request, $id)
    {
        if(Auth::user()->is_admin == 1){
            $post = Task::findOrFail($id);
            $post->title = $request->input('title');
            $post->body = $request->input('body');
            $post->save();
            if($post){
             return redirect('tasks');
            }
        }
    }
```

Here, you may ask, how does Laravel know the wildcard reference or input variable `$id` along with the `Request $request` object? This happens through the `Reflection` class that Laravel uses to guess what you are type hinting, and this is called *method injection*.

Basically, you could have passed the parameter's `Model` object as `'` instead of `$id`; however, since through the edit form you have passed `$id`, Laravel resolves the dependencies and injects it into the controller instance in place of `'`.

The position of the `$id` along with the `Request $request` object is interchangeable. Laravel is smart enough to read the wildcard entry and the `$request` object here. So, you can type hint the dependencies on your controller methods. The most common use case is the injection of the `Illuminate\Http\Request` instance.

If your controller method expects input from a `route` parameter, as happens in the update case where you want to catch the `$id` of the item you want to edit, behind the resource controller, you have a route like this:

```
Route::put('task/{id}', 'TaskController@update');
```

You can also type hint any dependencies your controller may need in the constructor. The declared dependencies will automatically be resolved and injected into the controller instance. Suppose you have a user repository in the app/Repositories folder like this:

```
//code 3.22
//app/Repositories/DBUserRepository.php
<?php namespace RepositoryDB;
use RepositoryInterface\UserRepositoryInterface as UserRepositoryInterface;
use App\User;
use Illuminate\Http\Request;

class DBUserRepository implements UserRepositoryInterface {

    public function all() {
        return User::all();
    }
}
```

You also have the app/Repositories/Interfaces folder, where you have your respective interface that looks like this:

```
//code 3.23
//app/Repositories/Interfaces/UserRepositoryInterface.php
<?php namespace RepositoryInterface;

 interface UserRepositoryInterface {
     public function all();
}
```

Now you can inject your dependencies into your UserController constructor quite easily. I have type hinted the contract (here UserRepositoryInterface), and the Laravel container has resolved it successfully.

```
//code 3.24
//app/HTTP/Controllers/UserController.php
class UserController extends Controller {
    public $users;
    public function __construct(DBUserRepository $users) {
        $this->users = $users;
    }
...//code continues
}
```

The instance of DBUserRepository always provides better testability.

How a Blade Template Works with Controllers and Models

Blade is a simple yet powerful template engine. It reduces the workload of the front-end staff drastically. With Blade templates, you can also use PHP code. Besides, it has its own PHP syntax, which is extremely easy to use.

The advantage of Blade views is that they come with automatic caching. All Blade view pages are compiled into plain PHP code and cached until you modify them. This means zero overhead for the application.

You can create any Blade view page with a `.blade.php` extension, and they should be stored inside the `resources/views` folder. For a large application, you may create separate folders for each resource and keep your view pages inside that. Likewise, I have kept my `Task` resource view pages inside the `resources/views/tasks` folder.

Two basic benefits of Blade are template inheritance and sections.

For the `Task` resource, you are using the default `resources/views/layouts/app.blade.php` file here. So, the `resources/views/tasks/index.blade.php` file is a simple view page that outputs the tasks.

```
//code 3.25
//resources/views/tasks/index.blade.php
@extends('layouts.app')
@section('content')
<div class="container">
    <div class="row justify-content-center">
        <div class="col-md-8">
            <div class="card">
                <div class="card-header">Task Page</div>
                <div class="card-body">
                    @foreach($tasks as $task)
                    <a href="/tasks/{{ $task->id }}">
                        {{ $task->title }}
                    </a>
                    <p></p>
                    @endforeach
                </div>
            </div>
        </div>
    </div>
</div>
@endsection
```

As you see, you can easily connect your view page to the master layout page by writing the following on the top of the page: `@extends('layouts.app')`. In between `@section('content')` and `@endsection`, you insert the content part. Basically, you are following the DOM object model for displaying the HTML page. It starts on the top and flows down to the end.

The master layout Blade page should have the header and the footer. If there is any fixed sidebar, according to the Bootstrap theme, you can include that also. If you create a simple master layout Blade page, it will look like the `app.blade.php` file in your `resources/views/layouts` folder, as shown here:

```
//code 3.26
    <html>
        <head>
            <title>App Name - @yield('title')</title>
        </head>
        <body>
            <div class="container">
                @yield('content')
            </div>
        </body>
    </html>
```

The `@yield('content')` part uses your `index.blade.php` page content. In the default `app.blade.php` page, that part is included like this:

```
<main class="py-4">
            @yield('content')
        </main>
```

But I have overwritten the default `app.blade.php` page to make it simple to understand. When Laravel installs, it comes with a default `app.blade.php` view page in the `resources/views/layouts` folder.

The `@section` and `@yield` directives are there with the typical HTML markup. The `@section` directive defines a section of content. Here, the `@yield` directive is used to display the contents of a given section.

Security in Blade

In any Blade view page, you usually get the data inside curly braces: `{{ $data->body }}'`. These statements are automatically sent through PHP's `htmlspecialchars` function to avoid XSS attacks so that the data is escaped.

If you do not want to escape data, then use this syntax: `{!! $data->body !!}`. But, be careful about the user's input. It is always a good practice to escape data supplied by users. The user's input data should be displayed using double curly braces: `{{ $user->data }}'`.

Authentication Through Blade

You can check whether the user is an administrator in the Blade view page, as shown here:

```
//code 3.27
    @foreach ($users as $user)
        @if ($user->admin == 1)
            @continue
        @endif

        <li>{{ $user->name }}</li>
        @if ($user->number == 1)
            @break
        @endif
    @endforeach
```

You can also check whether the user is a registered member with this simple method:

```
//code 3.28
    @auth
        // The user is authenticated...
    @endauth

    @guest
        // The user is not authenticated...
    @endguest
```

In Chapter 8, I will discuss the guard; however, in the Blade view page, you can use it directly like this:

```
//code 3.29
    @auth('admin')
        // The user is authenticated...
    @endauth

    @guest('admin')
        // The user is not authenticated...
    @endguest
```

Control Structures in Blade

You have already seen how you can manage looping through a Blade template; you can put control structures in any Blade view page like this:

```
//code 3.30
@if (count($users) === 1)
        I have one user!
    @elseif (count($users) > 1)
        I have multiple users!
    @else
        I don't have any users!
    @endif
```

Take a look at the following example where the @unless directive makes authentication much simpler:

```
//code 3.31
    @unless (Auth::check())
        You are not signed in.
    @endunless
```

Other Advantages of Blade Templates

You can also use the PHP isset function in a more convenient way like this:

```
//code 3.32
    @isset($images)
        // $images is defined and is not null...
    @endisset

    @empty($images)
        // $images is 'empty'...
    @endempty
```

You have already seen how you can use the foreach loop for getting records straight out of the database tables; in addition, you can restrict the number of records in the view page instead of doing this inside the controller. You can even choose which records to show.

```
//code 3.33
@foreach ($users as $user)
      @if ($user->id == 1)
          @continue
      @endif

      <li>{{ $user->name }}</li>

      @if ($user->id == 3)
          @break
      @endif
  @endforeach
```

You have four registered users. The output is as follows:

```
//output of code 3.34
•    ss
•    admin
```

In this case, when the loop finds that the ID is 1, it continues. After that, when the code reaches the second user, the loop stops and spits out the first two users' names, and after that it stops and breaks out from the loop when it finds the third user.

You can manipulate your database records by managing the control structures in the Blade view pages. And in such cases, Laravel allows you to do that without touching the models and controllers. This is a big advantage of using Laravel.

Working with Models

The *model* is the most critical part of any web application; you could even call it the "brain" of your application. A good web application will be model-centric, with each resource being a representation of the model. In other words, the model sets the business logic for every resource, such as `user`, `task`, and so on.

In Laravel, you will usually use Eloquent ORM when building applications. Eloquent is a simple yet beautiful ActiveRecord implementation for working with a database. Each database table has a corresponding model, and without this model, you cannot query the data in the database table. Not only that, but the model plays a pivotal role in building relations between different tables.

For this reason, a model is also called an *Eloquent model*, and you use a flag like `--migration`, or simply `--m`, while creating a model through the terminal. In the previous chapter, you saw how to create a `Task` model along with the `tasks` table. You also have used a `TaskController` to manage the `task` resource. In the next sections, you will see how you can create `tasks`, and you will also see how you can use route model binding to edit your `task` resource.

Route Model Binding: Custom and Implicit

A model is of no use if you don't have any underlying table associated with it. In a larger sense, you will often have a database query related to a model representing the resource. A query, whether it handles all records of a table or a single record, always points to one or many IDs. Whenever you perform a database operation, you actually do it through the model. Let's consider the `TaskController show` method to display the specified resource, as shown here:

```
//code 4.1
//app/HTTP/Controllers/TaskController.php
    public function show($id)
```

S. Sinha, *Beginning Laravel*, https://doi.org/10.1007/978-1-4842-4991-8_4

```
    {
        $task = Task::findOrFail($id);
        return view('tasks.show', compact('task'));
    }
```

Here you inject a model ID to this controller action to retrieve the model that corresponds to that ID.

Now if you issue the `php artisan route:list` command, you will see the following output for this particular action:

```
GET|HEAD  | tasks/{task}                | tasks.show        | App\Http\
Controllers\TaskController@show
```

In the previous output, you can see that `Method` is `GET`, `URI` is `tasks/{task}`, and `Name` is `tasks.show`. This means the view Blade page is `show.blade.php`, and it belongs to the `tasks` folder. Finally, `Action` is `App\Http\Controllers\TaskController@show`.

This means Laravel has automatically injected the model instances directly into the routes. The route lists show that you can inject an entire `Task` model instance that matches the given ID.

This route model binding is best understood when you use the form template in `edit.blade.php` belonging to the `resources/views/tasks` folder, as shown here:

```
//code 4.2
//resources/views/tasks/edit.blade.php
<div class="card-body">
{!! Form::model($task, ['route' => ['tasks.update', $task->id], 'method' =>
'PUT']) !!}

<div class="form-group">
{!! Form::label('title', 'Title', ['class' => 'awesome']) !!}
{!! Form::text('title', $task->title, ['class' => 'form-control']) !!}
</div>
<div class="form-group">
{!! Form::label('body', 'Body', ['class' => 'awesome']) !!}
{!! Form::textarea('body', $task->body, ['class' => 'form-control']) !!}
</div>
<div class="form-group">
```

```
{!! Form::submit('Edit Task', ['class' => 'btn btn-primary form-control'])
!!}
</div>
{!! Form::close() !!}
</div>
```

In the previous code, this line is important:

```
{!! Form::model($task, ['route' => ['tasks.update', $task->id], 'method' =>
'PUT']) !!}
```

Why does the route point to `tasks.update`, instead of `tasks.edit`? You can understand it better if you look at `route:list`, which tells you the action verb or method is `tasks.update`. Also, it points to the action `TaskController@update`.

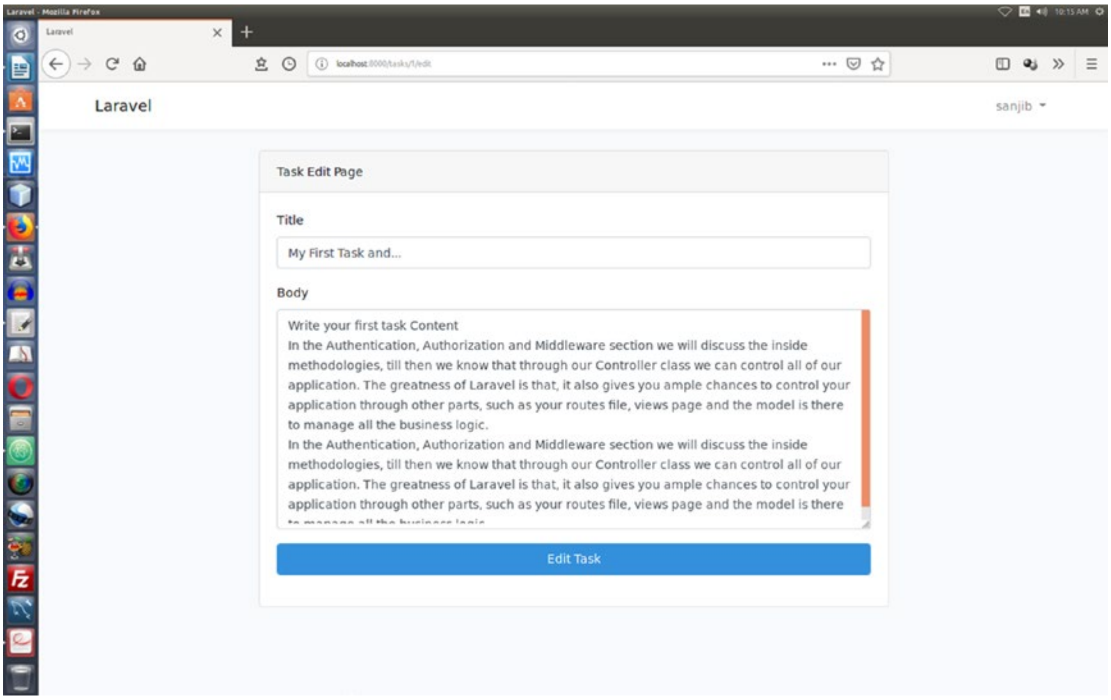
This is part of the RESTful action on the resource model `Task`. What is happening here is that the generated controller will already have methods stubbed for each of these actions. Laravel is smart enough to include notes in its service container, informing you which URIs and verbs they handle. So, you don't have to create the view page `tasks.update` in the `resource/views/tasks` folder. Through the RESTful resource controller, Laravel handles it.

You bind the route with the model `Task`. To understand this, you need to watch the `edit` method of `TaskController`.

```
//code 4.3
//app/HTTP/Controllers/TaskController.php
    public function edit($id)
    {
      if(Auth::user()->is_admin == 1){
        $task = Task::findOrFail($id);
        return view('tasks.edit', compact('task'));
      }
      else {
        return redirect('home');
      }
    }
```

Here you pass the `task` object as `$task` to `tasks.edit`, and Laravel is smart enough to guess that it is an instance of the `Task` model.

Now if you want to edit the first task that has an ID of 1, the URI is `http://localhost:8000/tasks/1/edit`, as shown in Figure 4-1.



Task Edit Page

Title

My First Task and...

Body

Write your first task Content
In the Authentication, Authorization and Middleware section we will discuss the inside methodologies, till then we know that through our Controller class we can control all of our application. The greatness of Laravel is that, it also gives you ample chances to control your application through other parts, such as your routes file, views page and the model is there to manage all the business logic.
In the Authentication, Authorization and Middleware section we will discuss the inside methodologies, till then we know that through our Controller class we can control all of our application. The greatness of Laravel is that, it also gives you ample chances to control your application through other parts, such as your routes file, views page and the model is there

Figure 4-1. *Route model binding through form*

Implicit Route Binding

You can achieve the same route model binding effect with implicit binding. In the previous code, you saw how to get the ID of the `task` resource. Now go ahead and change the code to the following:

```
//code 4.4
//app/HTTP/Controllers/TaskController.php
    public function edit(Task $task)
    {
      if(Auth::user()->is_admin == 1){
        return view('tasks.edit', compact('task'));
      }
```

```
        else {
          return redirect('home');
        }
      }
```

Laravel automatically resolves Eloquent models defined in routes or controller actions, as in `public function edit(Task $task){}`, whose type-hinted variable names match a route segment name. Since the `$task` variable is type-hinted as the `App\Task` Eloquent model and the variable name matches the `{task}` URI segment, Laravel will automatically inject the model instance that has an ID matching the corresponding value from the request URI.

Now if you want to edit the task with an ID of 2, the URI is `http://localhost:8000/tasks/2/edit`, and you get the same effect.

Custom Route Binding

By the way, you can customize the key name also. Suppose you have another database column other than ID and that database column is named `slug`.

In that case, you can override the `getRouteKeyName` method on the Eloquent model in the following way:

```
//code 4.5
    /**
     * Get the route key for the model.
     *
     * @return string
     */
    public function getRouteKeyName()
    {
        return 'slug';
    }
```

Now instead of the ID, you can pass `slug` as an instance of the `task` resource like before.

Model Relations

An application in Laravel mainly depends on model relations. Therefore, this section is one of the most important sections in the book. Up to now, you were trying to understand how the Laravel Model-View-Controller logic works. Now the time has come to build a small article management application. While building this application, you will learn about some other key components of Laravel. In the next chapter, I will go deeper into the database and Eloquent ORM components, and you will see how these relational operations work through models and how controllers control the workflow between models and views.

Building models from scratch is, no doubt, a cumbersome job, so Laravel has taken care of this in an elegant way. By default, Laravel comes with only the `User` model, but you can configure it according to your requirements. In most applications, however complicated they are, you do not have to touch the default `User` model installed by Laravel. You can add any extra columns in the `users` database table. According to the MVC pattern, the model talks to the database, either retrieves data from the database or inserts it, updates the data, and passes it to the controller. Then, the controller can pass it to the views. Conversely, the controller takes input from the views and passes it to the model to process data.

The `User` model interacts with the `users` database, and in the same way, other models will work in tandem with the other tables to run the article management application smoothly. In this application, you have many separate resources, such as articles, users, profiles, comments, and so on. These resources will interact with each other through models. Laravel has made the process super simple and expressive.

Behind the scenes, Laravel handles a huge task, which is evident from these simple expressive functions. Consider a few scenarios where a user has many comments. It is obvious that a user should be able to comment as much as they want. They should also be able to comment on another user's profile. So, the `comment` object may have a polymorphic relationship with two model objects: `Article` and `Profile`. One user may have many articles too. Now, each article may have many tags, and many tags may belong to many articles.

Therefore, you can see many types of relations here: article to user and the inverse (one-to-one), one user to one profile (one-to-one), one article to many tags (one-to-many), many articles to many tags (many-to-many), and comments to articles and profiles (polymorphic).

Note A *polymorphic* relationship is where a model can belong to more than one other model on a single association, such as comments belonging to the `articles` and `profiles` table records at the same time.

Before taking a look at the models, you must perform your migrations, covered next.

How Migrations Work with the Laravel Model

For the sake of building the example application, let's start with the database. I could have started with controllers and views or I could have discussed the model, but database migration is a good option because you are going to see how to create a completely database-driven application. This migration has a direct relation with the models.

You will see how it works in a minute.

Laravel supports four databases currently: MySQL, PostgreSQL, SQLite, and Microsoft SQL Server. The `config/database.php` file defines MySQL as the default database connection.

```
'default' => env('DB_CONNECTION', 'mysql'),
```

You can change this to match your requirements. Next, you will take a look at the database connection setup in the `config/database.php` file so that you have a proper understanding of how Laravel has designed this beautifully, as shown here:

```
'connections' => [

        'sqlite' => [
            'driver' => 'sqlite',
            'database' => env('DB_DATABASE', database_path('database.
            sqlite')),
            'prefix' => '',
        ],

        'mysql' => [
            'driver' => 'mysql',
            'host' => env('DB_HOST', '127.0.0.1'),
            'port' => env('DB_PORT', '3306'),
            'database' => env('DB_DATABASE', 'forge'),
            'username' => env('DB_USERNAME', 'forge'),
            'password' => env('DB_PASSWORD', ''),
            'unix_socket' => env('DB_SOCKET', ''),
            'charset' => 'utf8mb4',
            'collation' => 'utf8mb4_unicode_ci',
            'prefix' => '',
            'strict' => true,
            'engine' => null,
        ],

        'pgsql' => [
            'driver' => 'pgsql',
            'host' => env('DB_HOST', '127.0.0.1'),
            'port' => env('DB_PORT', '5432'),
            'database' => env('DB_DATABASE', 'forge'),
            'username' => env('DB_USERNAME', 'forge'),
            'password' => env('DB_PASSWORD', ''),
            'charset' => 'utf8',
            'prefix' => '',
            'schema' => 'public',
            'sslmode' => 'prefer',
        ],

        'sqlsrv' => [
            'driver' => 'sqlsrv',
            'host' => env('DB_HOST', 'localhost'),
            'port' => env('DB_PORT', '1433'),
            'database' => env('DB_DATABASE', 'forge'),
            'username' => env('DB_USERNAME', 'forge'),
            'password' => env('DB_PASSWORD', ''),
            'charset' => 'utf8',
            'prefix' => '',
        ],

    ],
```

Here, I would like to add one key concept: object-relational mapping (ORM). As an intermediate PHP user, you are probably already acquainted with this concept; yet, a few lines about it won't hurt you, as Laravel has managed this feature magnificently via Eloquent ORM. Mapping the application object to the database tables has been one of the greatest challenges of using Laravel; however, Eloquent ORM has solved this problem completely. It provides an interface that converts application objects to database table records and does the reverse too.

The next line in the file is also important:

```
'migrations' => 'migrations',
```

What does this mean? It is a default name of the table used for the project's migration status. Don't change it.

The next lines of code look like this:

```
'redis' => [

        'client' => 'predis',

        'default' => [
            'host' => env('REDIS_HOST', '127.0.0.1'),
            'password' => env('REDIS_PASSWORD', null),
            'port' => env('REDIS_PORT', 6379),
            'database' => 0,
        ],

    ],
```

Redis is an open source, fast, key-value store; among other things, you can manage cache and session data with it.

After studying the `config/database.php` file, you will look at the `.env` file. It's an important file that plays a major role in building an awesome application. You will find it in the root directory. When you build a database-driven application, you can locally create a database called `laravelmodelrelations` in MySQL. The environment file `.env` just points it to the database name, including your MySQL username and password. You are interested in these lines:

```
DB_DATABASE=laravelmodelrelations
DB_USERNAME=root
DB_PASSWORD=********
```

By default, the database is homestead; you can use this one, or you can change it accordingly. I have created a database called laravelmodelrelations and connected it with my Laravel application through this .env file.

Now, using migration, I can build my database tables and relate them with each other using some simple default methodology.

Keeping your application logic in your mind, you need to design the tables. Laravel comes up with two tables: users and password-resets. You do not need to change the second one. However, if needed, you can change the users table. Here, you don't do that, because you will have a separate Profile table where you will add some more information about a user. Besides, you need to build other tables as your application demands.

Let's see the database/migrations/create_password_resets_table.php file first.

```
// code 4.6
//  database/migrations/create_password_resets_table.php
public function up()
    {
        Schema::create('password_resets', function (Blueprint $table) {
            $table->string('email')->index();
            $table->string('token');
            $table->timestamp('created_at')->nullable();
        });
    }
public function down()
    {
        Schema::dropIfExists('password_resets');
    }
```

I didn't paste the whole code here for brevity. Let me explain the first chunk: through the up() method, the Schema class creates the table password_resets, and it is created dynamically in your database using a closure, where the Blueprint table object takes charge. Inside this function, you can add extra functionalities. In database/migrations/create_password_resets_table.php, you don't have to do that. Next, you will build the Profile model and table by issuing a single command, as shown here:

```
// code 4.7
$ php artisan make:model Profile -m
```

```
Model created successfully.
Created Migration: 2018_09_16_235301_create_profiles_table
```

The first model, called `Profile`, has been created under the app folder. Next, you should work on the `profiles` table before running the migration.

The `profiles` table has the following code inside `database/migrations` folder; the file name is generated by Laravel, prefixed by the data in which it is generated:

```
// code 4.8
//profiles table comes up by default

public function up()
    {
        Schema::create('profiles', function (Blueprint $table) {
            $table->increments('id');
            $table->timestamps();
        });
    }
```

In fact, whenever you create a model with the respective table, it already has a skeleton like this. So, for the next tables, I won't repeat this skeleton anymore. You need to add some more functionality in this code. So, it looks like this:

```
// code 4.9
public function up()
    {
        Schema::create('profiles', function (Blueprint $table) {
            $table->increments('id');
            $table->integer('user_id');
            $table->string('city', 64);
            $table->text('about');
            $table->timestamps();
        });
    }
```

I could have added more functionalities such as `first name`, `last name`, and so on, but I didn't do that for brevity. Currently, our purposes will be served by only `city` and `about`. The interesting part is of course `user_id`. Each profile is connected with a single user.

Therefore, you can say that each user has one profile, and one profile belongs to one user. You can define that relationship in two respective models: `User` and `Profile`. I will cover that relationship later; right now let's create other models and tables by issuing these commands one after another:

```
// code 4.10
$ php artisan make:model Profile -m
Model created successfully.
Created Migration: 2018_09_17_233619_create_profiles_table
$ php artisan make:model Tag -m
Model created successfully.
Created Migration: 2018_09_18_013602_create_tags_table
$ php artisan make:model Article -m
Model created successfully.
Created Migration: 2018_09_18_013613_create_articles_table
$ php artisan make:model Comment -m
Model created successfully.
Created Migration: 2018_09_18_013629_create_comments_table

$ php artisan make:migration create_article_tag_table --create=article_tag
Created Migration: 2018_09_17_094743_create_article_tag_table
```

Let me explain what you are doing here. By using a single command like `php artisan make:model Comment -m`, you create a model comment, and at the same time Laravel creates a related `comments` table for it.

Now, before running the migration, you should plan your application and add functionalities to your newly created tables based on code 4.9.

You have the `users` and `profiles` tables, and you have a corresponding key that may attach one profile to one user. So, you will add these methods to the `User` and `Profile` models, respectively.

```
// code 4.11
// app/User.php
public function profile() {

        return $this->hasOne('App\Profile');

    }
```

This means one user has one profile, and it attaches to the `App\Profile` model. This is a one-to-one relationship between the `User` and `Profile` models. Now it has an inverse in the `Profile` model.

```
// code 4.12
// app/Profile.php
public function user() {

        return $this->belongsTo('App\User');
    }
```

This code does the same thing, only in an inverse way. Each profile should belong to one `User` model. To speed up your application, let's add functionalities to other tables and after that define the relationship accordingly.

Next, you have the `tags` and `articles` tables. They have an interesting relationship between them. Many tags belong to many articles, and the inverse is also true. With some simple logic, you can say they have a many-to-many relation. So, you need a pivot table to maintain that relationship. Let's see the functionalities of those tables first, shown here:

```
// code 4.13
// tags table
public function up()
    {
        Schema::create('tags', function (Blueprint $table) {
            $table->increments('id');
            $table->string('tag');
            $table->timestamps();
        });
    }
```

At the same time, I mentioned the relationship with `article` at the `Tag` model, as follows:

```
//Tag.php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
```

```
class Tag extends Model
{
    //
    protected $fillable = [
        'tag'
    ];

    public function articles() {

        return $this->belongsToMany('App\Articles');

    }
}
```

Next you have `articles` table, as shown here:

```
// code 4.14
// articles table
public function up()
    {
        Schema::create('articles', function (Blueprint $table) {
            $table->increments('id');
            $table->integer('user_id');
            $table->string('title');
            $table->text('body');
            $table->timestamps();
        });
    }
```

I will also mention the relationship with other resource models and related tables including Tag, as shown here:

```
//Article.php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
```

```
class Article extends Model
{
    //
    protected $fillable = [
        'user_id', 'title', 'body',
    ];

    public function user() {
        return $this->belongsTo('App\User');
    }

    /**
     * Get the tags for the article
    */

   public function tags() {
       return $this->belongsToMany('App\Tag');
   }

   /**
   * Get all of the profiles' comments.
   */
   public function comments(){
     return $this->morphMany('App\Comment', 'commentable');
   }
}
```

Also, now you have the pivot table `article_tag`, as shown here:

```
// code 4.15
// article_tag table
public function up()
    {
        Schema::create('article_tag', function (Blueprint $table) {
            $table->increments('id');
            $table->integer('article_id');
            $table->integer('tag_id');
            $table->timestamps();
        });
    }
```

You'll also want to create a Role model and roles table because you want your users to have some certain privileges such as Administrator, Moderator, and Member. Therefore, you need a pivot table like role_user.

Let's create the roles. The commands are simple, as shown here:

```
$ php artisan make:model Role -m
```

To create a role_user table, you issue this command:

```
$ php artisan make:migration create_role_user_table –create=role_user
```

The Role model has some methods that allow it to have a relationship with the User model. Let's take a look at the code:

```
// app/Role.php
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
class Role extends Model
{
    protected $fillable = [
      'name'
  ];
  public function user() {
      return $this->belongsTo('App\User');
  }
  public function users() {
      return $this->belongsToMany('App\User');
  }
}
```

One role may belong to one user, and at the same time it may belong to many users. So, you need to add two methods in your User model, as shown here:

```
//app.User.php
public function role() {
        return $this->belongsTo('App\Role');
    }
    public function roles() {
        return $this->belongsToMany('App\Role');
    }
```

One user may have one role or many roles at the same time.

Now you need to work on the tables you have just created. First, add a name column to the roles table, as shown here:

```
// database/migrations/roles table
public function up()
    {
        Schema::create('roles', function (Blueprint $table) {
            $table->increments('id');
            $table->string('name');
            $table->timestamps();
        });
    }
```

At the same time in your pivot table role_user, add two columns, as shown here: role_id and user_id.

```
// database/migrations/roles table
public function up()
    {
        Schema::create('role_user', function (Blueprint $table) {
            $table->increments('id');
            $table->integer('role_id');
            $table->integer('user_id');
            $table->timestamps();
        });
    }
```

I have not touched the comment table currently, for one reason. You will work on that table at the end phase of building your content management application. This is because the comment table will have polymorphic relations with other tables. I will discuss it in the next chapter.

So, you have most of the tables ready to migrate from Laravel to your database. You can run the migration by using this command:

```
// code 4.16
$ php artisan migrate
Migrating: 2018_09_17_233619_create_profiles_table
Migrated:  2018_09_17_233619_create_profiles_table
Migrating: 2018_09_18_013602_create_tags_table
Migrated:  2018_09_18_013602_create_tags_table
Migrating: 2018_09_18_013613_create_articles_table
Migrated:  2018_09_18_013613_create_articles_table
Migrating: 2018_09_18_013629_create_comments_table
Migrated:  2018_09_18_013629_create_comments_table
Migrating: 2018_09_18_013956_create_article_tag_table
Migrated:  2018_09_18_013956_create_article_tag_table
...
the list is incomplete
```

Since you have migrated your tables, you can now populate them with fake data. It can be a tedious job to add fake data manually. Laravel has taken care of that so you are able to test your application with fake data.

Model and Faker Object

To populate the database with fake data, open your database/factory/UserFactory.php file. You will use the Faker object to define the model and add some fake data into it.

```
// code 4.17
//  database/factory/UserFactory.php
$factory->define(App\User::class, function (Faker $faker) {
    return [
        'name' => $faker->name,
        'email' => $faker->unique()->safeEmail,
        'password' => '$2y$10$TKh8H1.PfQx37YgCzwiKb.
        KjNyWgaHb9cbcoQgdIVFlYg7B77UdFm', // secret
        'remember_token' => str_random(10),
    ];
});

$factory->define(App\Article::class, function (Faker $faker) {
    return [
        'user_id' => App\User::all()->random()->id,
        'title' => $faker->sentence,
        'body' => $faker->paragraph(random_int(3, 5))
    ];
});

$factory->define(App\Profile::class, function (Faker $faker) {
    return [
        'user_id' => App\User::all()->random()->id,
        'city' => $faker->city,
        'about' => $faker->paragraph(random_int(3, 5))
    ];
});

$factory->define(App\Tag::class, function (Faker $faker) {
    return [
        'tag' => $faker->word
    ];
});
$factory->define(App\Role::class, function (Faker $faker) {
    return [
        'name' => $faker->word
    ];
});
```

Let me explain this code a little bit. Here, you are using the `faker` object to populate the database tables with some dummy data. You will run the database seeds through `database/seeds/DatabaseSeeder.php`. The code looks like this:

```
// code 4.18
// database/seeds/DatabaseSeeder.php
public function run()
    {
        factory(App\User::class, 10)->create()->each(function($user){
            $user->profile()->save(factory(App\Profile::class)->make());
        });

        factory(App\Tag::class, 20)->create();

        factory(App\Article::class, 50)->create()->each(function($article){
          $ids = range(1, 50);
          shuffle($ids);
          $sliced = array_slice($ids, 1, 20);
          $article->tags()->attach($sliced);
        });

        factory(App\Role::class, 3)->create()->each(function($role){
          $ids = range(1, 5);
          shuffle($ids);
          $sliced = array_slice($ids, 1, 20);
          $role->users()->attach($sliced);
        });

    }
```

You are going to create 10 users, 20 tags, and 50 articles. One single command will handle the whole operation. In the last two `factory()` methods, you attached `tags` with the `article` object, and `role` with `users`. While populating the tables with fake data, it will automatically attach some tags with some articles. The same is true for roles and users.

```
// code 4.19
$ php artisan migrate:refresh -seed
```

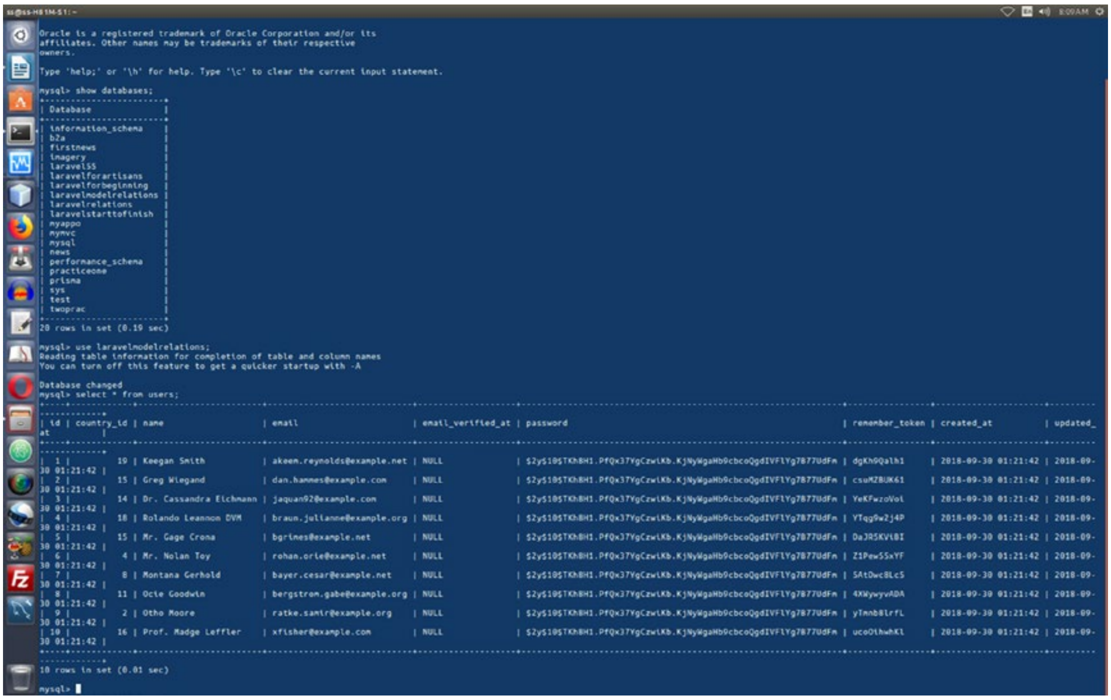
This will give you some long output where it rolls back all the tables and migrates them instead. Remember, any time you issue this command, it will populate the database tables with a new combination of data. This enhances the testability, because you can always add some new columns in a table and run this command, and it will roll back the old tables and migrate the new.

```
// output
Rolling back: 2018_09_18_013956_create_article_tag_table
Rolled back:  2018_09_18_013956_create_article_tag_table
Rolling back: 2018_09_18_013629_create_comments_table
Rolled back:  2018_09_18_013629_create_comments_table
Rolling back: 2018_09_18_013613_create_articles_table
Rolled back:  2018_09_18_013613_create_articles_table
Rolling back: 2018_09_18_013602_create_tags_table
Rolled back:  2018_09_18_013602_create_tags_table
Rolling back: 2018_09_17_233619_create_profiles_table
Rolled back:  2018_09_17_233619_create_profiles_table
Rolling back: 2014_10_12_100000_create_password_resets_table
Rolled back:  2014_10_12_100000_create_password_resets_table
Rolling back: 2014_10_12_000000_create_users_table
Rolled back:  2014_10_12_000000_create_users_table
Migrating: 2014_10_12_000000_create_users_table
Migrated:  2014_10_12_000000_create_users_table
Migrating: 2014_10_12_100000_create_password_resets_table
Migrated:  2014_10_12_100000_create_password_resets_table
Migrating: 2018_09_17_233619_create_profiles_table
Migrated:  2018_09_17_233619_create_profiles_table
Migrating: 2018_09_18_013602_create_tags_table
Migrated:  2018_09_18_013602_create_tags_table
Migrating: 2018_09_18_013613_create_articles_table
Migrated:  2018_09_18_013613_create_articles_table
Migrating: 2018_09_18_013629_create_comments_table
Migrated:  2018_09_18_013629_create_comments_table
Migrating: 2018_09_18_013956_create_article_tag_table
Migrated:  2018_09_18_013956_create_article_tag_table
...
this list is incomplete
```

To watch the real actions in your database tables, you don't have to use phpMyAdmin always. You can check it on your terminal also in this way:

```
// code 4.20
mysql> use laravelmodelrelations
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables;
+----------------------------------+
| Tables_in_laravelmodelrelations |
+----------------------------------+
|  article_tag                      |
| articles                         |
| comments                         |
| migrations                       |
| password_resets                  |
| profiles                         |
| role_user                        |
| roles                            |
| tags                             |
| users                            |
+----------------------------------+
10 rows in set (0.00 sec)

mysql> select * from users;
```

This will also give you a long listing of all ten users on your terminal, as shown in Figure 4-2.



```
ss@ss-H81M-S1: ~

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+-------------------------+
| Database                |
+-------------------------+
| information_schema      |
| b2a                     |
| firstnews               |
| imagery                 |
| laravel55               |
| laravelforartisans      |
| laravelforbeginning     |
| laravelmodelrelations   |
| laravelrelations        |
| laravelstarttofinish    |
| myappo                  |
| mymvc                   |
| mysql                   |
| news                    |
| performance_schema      |
| practiceone             |
| prisma                  |
| sys                     |
| test                    |
| twoprac                 |
+-------------------------+
20 rows in set (0.19 sec)

mysql> use laravelmodelrelations;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> select * from users;
+----+------------+-------------------------+-----------------------------+-------------------+--------------------------------------------------------------+----------------+---------------------+---------
-------------+
| id | country_id | name                    | email                       | email_verified_at | password                                                     | remember_token | created_at          | updated_
at           |
+----+------------+-------------------------+-----------------------------+-------------------+--------------------------------------------------------------+----------------+---------------------+---------
-------------+
|  1 |         19 | Keegan Smith            | akeem.reynolds@example.net  | NULL              | $2y$10$TKhBH1.PfQx37YgCzwlKb.KjNyWgaHb9cbcoQgdIVFlYg7B77UdFm | dgKh0Qalh1     | 2018-09-30 01:21:42 | 2018-09-
30 01:21:42 |
|  2 |         15 | Greg Wiegand            | dan.hammes@example.com      | NULL              | $2y$10$TKhBH1.PfQx37YgCzwlKb.KjNyWgaHb9cbcoQgdIVFlYg7B77UdFm | csuMZBUK61     | 2018-09-30 01:21:42 | 2018-09-
30 01:21:42 |
|  3 |         14 | Dr. Cassandra Eichmann  | jaquan92@example.com        | NULL              | $2y$10$TKhBH1.PfQx37YgCzwlKb.KjNyWgaHb9cbcoQgdIVFlYg7B77UdFm | YeKFwzoVoi     | 2018-09-30 01:21:42 | 2018-09-
30 01:21:42 |
|  4 |         18 | Rolando Leannon DVM     | braun.julianne@example.org  | NULL              | $2y$10$TKhBH1.PfQx37YgCzwlKb.KjNyWgaHb9cbcoQgdIVFlYg7B77UdFm | YTqg9w2j4P     | 2018-09-30 01:21:42 | 2018-09-
30 01:21:42 |
|  5 |         15 | Mr. Gage Crona          | bgrimes@example.net         | NULL              | $2y$10$TKhBH1.PfQx37YgCzwlKb.KjNyWgaHb9cbcoQgdIVFlYg7B77UdFm | DaJR5KVtBI     | 2018-09-30 01:21:42 | 2018-09-
30 01:21:42 |
|  6 |          4 | Mr. Nolan Toy           | rohan.orie@example.net      | NULL              | $2y$10$TKhBH1.PfQx37YgCzwlKb.KjNyWgaHb9cbcoQgdIVFlYg7B77UdFm | Z1Pew5SxYF     | 2018-09-30 01:21:42 | 2018-09-
30 01:21:42 |
|  7 |          8 | Montana Gerhold         | bayer.cesar@example.net     | NULL              | $2y$10$TKhBH1.PfQx37YgCzwlKb.KjNyWgaHb9cbcoQgdIVFlYg7B77UdFm | 5AtDwc8LcS     | 2018-09-30 01:21:42 | 2018-09-
30 01:21:42 |
|  8 |         11 | Ocie Goodwin            | bergstrom.gabe@example.org  | NULL              | $2y$10$TKhBH1.PfQx37YgCzwlKb.KjNyWgaHb9cbcoQgdIVFlYg7B77UdFm | 4XWywyvADA     | 2018-09-30 01:21:42 | 2018-09-
30 01:21:42 |
|  9 |          2 | Otho Moore              | ratke.samir@example.org     | NULL              | $2y$10$TKhBH1.PfQx37YgCzwlKb.KjNyWgaHb9cbcoQgdIVFlYg7B77UdFm | yTmnb8lrfL     | 2018-09-30 01:21:42 | 2018-09-
30 01:21:42 |
| 10 |         16 | Prof. Madge Leffler     | xfisher@example.com         | NULL              | $2y$10$TKhBH1.PfQx37YgCzwlKb.KjNyWgaHb9cbcoQgdIVFlYg7B77UdFm | ucoOihwhKl     | 2018-09-30 01:21:42 | 2018-09-
30 01:21:42 |
+----+------------+-------------------------+-----------------------------+-------------------+--------------------------------------------------------------+----------------+---------------------+---------
-------------+
10 rows in set (0.01 sec)

mysql> 
```

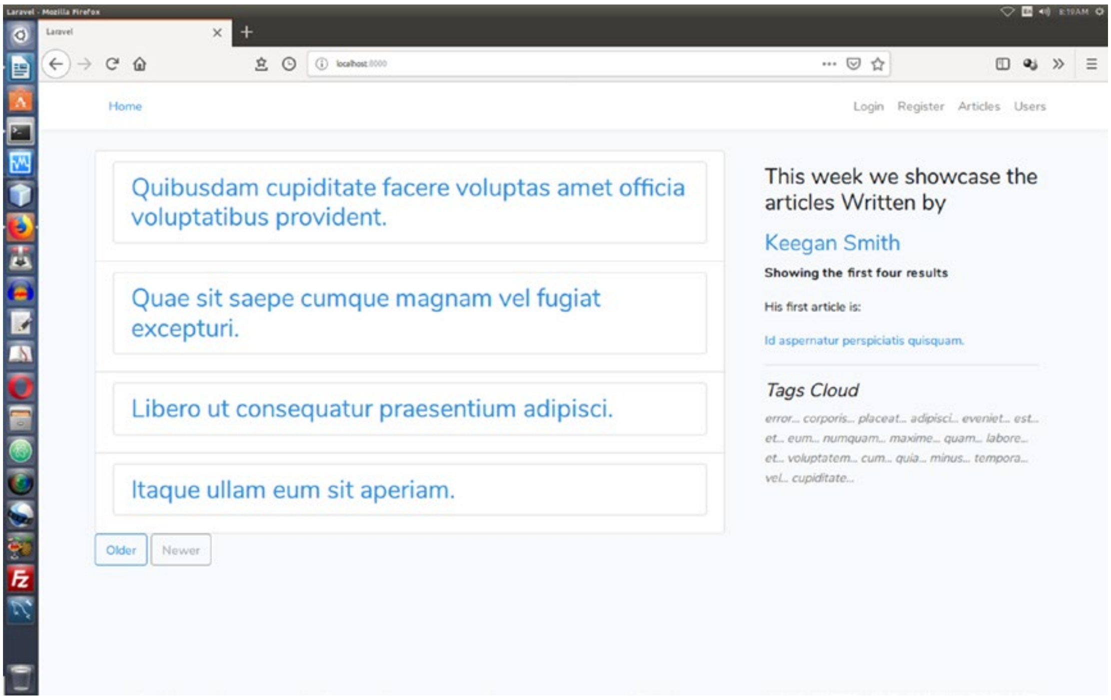
Figure 4-2. *All dummy users in the users table*

Take the third user; the user's name is `Van Gorczany`, and the e-mail ID is `kerluke.filomena@example.net`. Now through that dummy e-mail, you can log in to your application. Every user has one password, which is `secret`.

You have migrated all tables except `Comment`. You have dummy data inside the tables; therefore, you can now build the relations, and after that, you can use resource controllers and views to show how these model relations work perfectly.

Examining the Home Page

Before proceeding, let's take a look at what your home page will look like, as shown in Figure 4-3.



Quibusdam cupiditate facere voluptas amet officia voluptatibus provident.

Quae sit saepe cumque magnam vel fugiat excepturi.

Libero ut consequatur praesentium adipisci.

Itaque ullam eum sit aperiam.

This week we showcase the articles Written by

Keegan Smith

Showing the first four results

His first article is:

Id aspernatur perspiciatis quisquam.

Tags Cloud

error... corporis... placeat... adipisci... eveniet... est... et... eum... numquam... maxime... quam... labore... et... voluptatem... cum... quia... minus... tempora... vel... cupiditate...

***Figure 4-3.** The home page of the article management application*

To create the home page of your content management system (Figure 4-3), you need to slightly change the default `routes/web.php` page, and you can also change the code of the `resource/views/welcome.blade.php` file.

```
// code 4.21
//  routes/web.php
use App\Article;
use App\Tag;
Route::get('/', function () {
    $articles = Article::all();
    $tags = Tag::all();
    return view('welcome', ['articles' => $articles, 'tags' => $tags]);
});
```

All you have done here is to pass articles and tags separately. Now in the welcome.blade.php file, you will get the following output:

```
// code 4.22
// resources/views/welcome.blade.php
@extends('layouts.app')
@section('content')
<div class="container">
    <div class="row">
        <div class="col-md-8 blog-main col-lg-8 blog-main col-sm-8 blog-
        main">
          <div class="blog-post">
    <ul class="list-group">
         @foreach($articles as $article)
         <li class="list-group-item">
             <h2 class="blog-post-title">
                 <li class="list-group-item"><a href="/articles/{{
                 $article->id }}">{{ $article->title }}</a>
             </h2>
             Written by <a href="/users/{{ $article->user_id }}
             /articles">{{ $article->user->name }}</a>
         </li>
         @endforeach
    </ul>
            </div>
          <nav class="blog-pagination">
            <a class="btn btn-outline-primary" href="#">Older</a>
            <a class="btn btn-outline-secondary disabled" href="#">Newer</a>
          </nav>

        </div>
        <aside class="col-md-4 blog-sidebar">
          <div class="p-3">
              <h4 class="font-italic">Tags Cloud</h4>
            @foreach($tags as $tag)
            <font class="font-italic" color="gray"> {{ $tag->tag }}...
            @endforeach
          </div>
        </aside>
    </div>
</div>
@endsection
```

Let's look at this line from the previous code:

```
Written by <a href="/users/{{ $article->user_id }}/articles">{{ $article-
>user->name }}</a>
```

The `article` object directly accesses the `user` object and gets the respective name associated with that article. How does this happen? This is a good example of a one-to-one relationship. If you click the link, you will reach the user's page and can read all the articles written by that user. You need to understand one thing: every article belongs to one user, and you can access that user from that article. Laravel model relations handle that behind the scenes. To make them work properly, you have to define them in the models.

The next step will be to check all the models and finally define the relationship between them so that they can talk to each other to fetch the respective data. After that, you will build the resource controllers and views to complete the whole setup.

Next, you will see a series of code listings from your models so that you can associate these relational ideas with the models.

```
// code 4.23
// app/Article.php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Article extends Model
{
    protected $fillable = [
        'user_id', 'title', 'body',
    ];
    public function user() {
        return $this->belongsTo('App\User');
    }

    public function users() {
        return $this->belongsToMany('App\User');
    }

    public function tags() {
        return $this->belongsToMany('App\Tag');
    }
}
```

I have not included the comment methods here. I will do that as you progress through the book. Initially, the Article model has one-to-one, one-to-many, and many-to-many relations with the User and Tag models.

```
// code 4.24
// app/Profile.php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Profile extends Model
{
    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = [
        'user_id', 'city', 'about',
    ];

    public function user() {
        return $this->belongsTo('App\User');
    }
}
```

The `Profile` model has one method called `user` that defines the relationship with one `User`. You will see later how you can access the profile of any user from the `user` object.

```
// code 4.25
// app/Tag.php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Tag extends Model
{
    //
    protected $fillable = [
        'tag'
    ];

    public function articles() {

        return $this->belongsToMany('App\Articles');

    }
}
```

Next, you will take a look at the `User` model. It is important to note that the `User` model plays a vital role in this application.

```
// code 4.26
// app/User.php
<?php

namespace App;

use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use Notifiable;
    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = [
        'name', 'email', 'password',
    ];

    /**
     * The attributes that should be hidden for arrays.
     *
     * @var array
     */
    protected $hidden = [
        'password', 'remember_token',
    ];

    public function profile() {
        return $this->hasOne('App\Profile');
    }

    public function article() {
        return $this->hasOne('App\Article');
    }

    public function articles() {
        return $this->hasMany('App\Article');
    }
}
```

Currently, this model has one component missing, which is `Comment`. You will add it in the last stage. From the previous code, it is evident that the user has one profile, has one article, and many articles.

You have your database tables ready with dummy data. You have made your models ready to maintain the relationship among the objects. So, the time has come to make resource controllers. After that, you will build your views to test that your application works.

Relations Between Model, Database, and Eloquent

PHP database integration is a tedious process. It usually takes four essential steps to complete the database integration.

1. Laravel creates a connection.
2. Laravel selects a database; you know these parts, such as user name, password, etc have been defined in the `.env` file.
3. Usually, the third step is the most complicated one: performing a database query. Laravel's Eloquent ORM handles that complicated part in such a decent manner that you don't have to worry about it.
4. After that, Laravel automatically closes the connection. I have already discussed this topic in great detail.

There are some advantages to using a resource controller. With a single command, you can attach the controller to the associated model. Moreover, you can have every necessary method in one place. Let's create the resource `Comment` controller and associate it with the `Comment` model. A single command will do this, as shown here:

```
// code 4.27
$ php artisan make:controller CommentController --resource --model=Comment
```

Let's look at the newly created `Comment` controller. Now it is empty. However, it provides all the essential methods.

```
// code 4.28
// app/HTTP/Controllers/CommentController.php
<?php

namespace App\Http\Controllers;

use App\Comment;
use Illuminate\Http\Request;

class CommentController extends Controller
{

    public function index()
    {

    }
    public function create()
    {
        //
    }
    public function store(Request $request)
    {
        //
    }
    public function show(Comment $comment)
    {
        //
    }
    public function edit(Comment $comment)
    {
        //
    }
    public function update(Request $request, Comment $comment)
    {
        //
    }
    public function destroy(Comment $comment)
    {
        //
    }
}
```

The `index()` method will present the home page where you can show every comment in one place. Through the `show()` method, you can show each comment. While showing each comment, you can also show who the commenter is. The possibilities are endless. At the same time, by using this resource controller, you can create/store, edit/update, and finally destroy the `comment` object.

So, you create other resource controllers in the same way.

```
// code 4.29
$ php artisan make:controller UserController --resource -model=User
$ php artisan make:controller ArticleController --resource -model=Article
$ php artisan make:controller TagController --resource -model=Tag
```

Now you can define the routes in your `routes/web.php` file.

```
// code 4.30
//  routes/web.php
Auth::routes();
Route::get('/home', 'HomeController@index')->name('home');

Route::resource('users', 'UserController');
Route::resource('profiles', 'ProfileController');
Route::resource('articles', 'ArticleController');
Route::resource('comments', 'CommentController');
Route::get('/users/{id}/articles', 'ArticleController@articles');
Route::get('/', 'ArticleController@main');
```

I have also changed the main route. In the previous process, you used an anonymous function to return the view; now you are taking that view under the `main()` method of `ArticleController`.

Creating Views to Show Relationships

For the first phase of your application, you are ready to create the views. You have seen the `welcome.blade.php` page. To articles along with other associated objects, you are going to learn how to create three more view pages. They are `index.blade.php`, `show.blade.php`, and `articles.blade.php`.

Here's the code of the `index()` method in `ArticleController.php`:

```
// code 4.31
// app/HTTP/Controllers/ArticleController.php
    public function index()
    {
        $articles = Article::orderBy('created_at','desc')->paginate(4);
        $users = User::all();
        $tags = Tag::all();
        return view('articles.index', compact('articles', 'users', 'tags'));
    }
```

You are passing all articles with pagination links, users, and tags to the index page of all articles so that you can move to other pages easily. You can do that because of the Eloquent ORM, which provides you with a simple yet elegant `ActiveRecord` implementation for working with your database. Here you might have noticed that each database table has a corresponding model, and I have used that to get all the records, as follows:

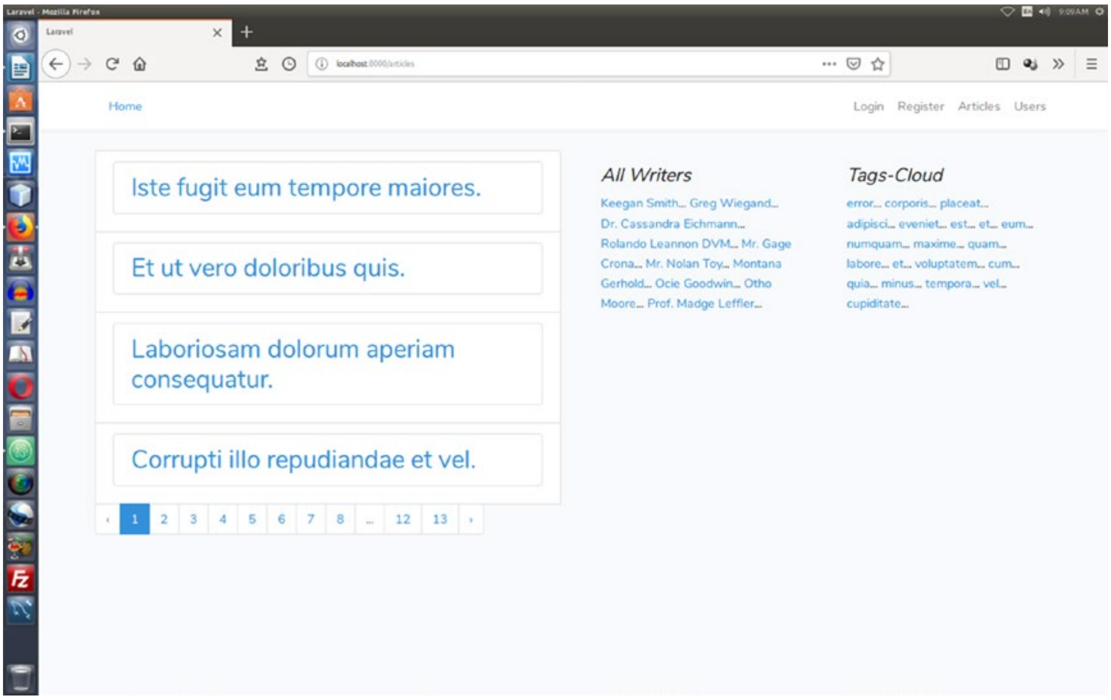
```
$articles = Article::orderBy('created_at,'desc')->paginate(4);
```

Here, the model `Article` allows you to query data in the tables, and you have just passed that data to the views with pagination links (here showing four articles per page). In the same way, later you will insert the data into the database. Next, you will get those values in the index page, so the code of `resources/views/articles/index.blade.php` looks like this:

```
// code 4.32
// resources/views/articles/index.blade.php
@extends('layouts.app')
@section('content')
<div class="container">
    <div class="row">
        <div class="col-md-6 blog-main col-lg-6 blog-main col-sm-6 blog-main">
            <div class="blog-post">
    <ul class="list-group">
         @foreach($articles as $article)
         <li class="list-group-item"><h2 class="blog-post-title">
                 <li class="list-group-item"><a href="/articles/{{
                 $article->id }}">{{ $article->title }}</a>

             </h2>
         </li>
        @endforeach
        {{ $articles->render() }}
    </ul>
           </div>
         <nav class="blog-pagination">

         </nav>
       </div>
       <aside class="col-md-3 blog-sidebar">
         <div class="p-3">
             <h4 class="font-italic">All Writers</h4>
           @foreach($users as $user)
                <a href="/users/{{ $user->id }}">{{ $user->name }}</a>...
        @endforeach
         </div>
       </aside>
       <aside class="col-md-3 blog-sidebar">
         <div class="p-3">
             <h4 class="font-italic">Tags-Cloud</h4>
           @foreach($tags as $tag)
                <a href="/tags/{{ $tag->id }}">{{ $tag->tag }}</a>...
        @endforeach
         </div>
       </aside>
    </div>
</div>
@endsection
```

You loop through the `articles` item and grab four articles on each page, and the code `{{ $articles->render() }}` gives you the pagination links.

Now, if you type `http://localhost:8000/articles` in your browser, it will show the page in Figure 4-4 with pagination in place.



Iste fugit eum tempore maiores.

Et ut vero doloribus quis.

Laboriosam dolorum aperiam consequatur.

Corrupti illo repudiandae et vel.

All Writers

Keegan Smith... Greg Wiegand... Dr. Cassandra Eichmann... Rolando Leannon DVM... Mr. Gage Crona... Mr. Nolan Toy... Montana Gerhold... Ocie Goodwin... Otho Moore... Prof. Madge Leffler...

Tags-Cloud

error... corporis... placeat... adipisci... eveniet... est... et... eum... numquam... maxime... quam... labore... et... voluptatem... cum... quia... minus... tempora... vel... cupiditate...

Figure 4-4. *The Articles page of the article management application*

This is where you find all the tables related to the respective models. You must understand the naming conventions that Eloquent ORM follows. When you create the model on the command line, Laravel creates a table for that. By convention, the snake_case plural name of the class has been used as the table name. You saw that for the `Article` model, a table `articles` has been created. You didn't tell Eloquent which table to use for your `Article` model. By default, it has chosen the name. First, in the `ArticleController` using the `index()` method, you start retrieving data from the database. Here the Eloquent model has acted as a powerful query builder allowing you to fluently query the database associated with the database.

Instead of writing this code in the `main()` method:

```
$articles = Article::all();
```

you can add constraints to queries and then use the `get` method to retrieve the results in this way:

```
$articles = Article::where('user_id', 1)->orderBy('title', 'desc')-
>take(5)->get();
```

This has brought a massive change in the whole structure of the main page at `http://localhost:8000`. You have used constraints to choose the `user_id` set to 1 and ordered the titles in descending order. The four records have been taken from the database.

Finally, your `main()` method of `ArticleController` looks like this:

```
// code 4.33
// app/HTTP/Controllers/ ArticleController.php
public function main()
    {
        $articles = Article::where('user_id', 1)->orderBy('title', 'desc')-
        >take(4)->get();
        $tags = Tag::all();
        return view('welcome', ['articles' => $articles, 'tags' => $tags]);
    }
```

The `welcome.blade.php` file now looks like this:

```
// code 4.34
//resources/views/ welcome.blade.php
<ul class="list-group">
         @foreach($articles as $article)
         <li class="list-group-item">
             <h2 class="blog-post-title">
                 <li class="list-group-item"><a href="/articles/
                 {{ $article->id }}">{{ $article->title }}</a>
             </h2>
         </li>
         @endforeach
    </ul>
...
<aside class="col-md-4 blog-sidebar">
          <div class="p-3">
              <h3 class="blog-post-title">This week you have showcased only
              the articles
                  Written by <a href="/users/{{ $article->user_id }}
                  /articles">{{ $article->user->name }}</a>
        </h3>
```

```
            <strong>Showing the first four results</strong>
            <hr class="linenums" color="red">
            <h4 class="font-italic">Tags Cloud</h4>
          @foreach($tags as $tag)
          <font class="font-italic" color="gray"> {{ $tag->tag }}...
          @endforeach
        </div>
      </aside>
```

You saw before how this logic looks in the home page (Figure 4-3). In the left sidebar panel, you are showcasing the articles by the user (with an ID of 1).

As you see, methods like `all()` and `get()` retrieve multiple results. Actually, an instance of `Illuminate\Database\Eloquent\Collection` has been returned. This `Collection` class provides many helpful methods for working with the Eloquent results. In this case, you can loop over the collection like an array.

However, the most interesting line in the code is as follows:

```
<h3 class="blog-post-title">This week we have showcased only the articles
                  Written by <a href="/users/{{ $article->user_id }}
                  /articles">{{ $article->user->name }}</a>
        </h3>
```

Why is this the most fascinating line of all? The `$article` object directly accesses the `$user` object and gets its `name` property. In the `ArticleController main()` method, you have directed the `Article` model to select the articles belonging to the user with an ID of 1. In addition, in the `Article` model, you are defining the method in this way:

```
// code 4.35
// app/Article.php
public function user() {
        return $this->belongsTo('App\User');
    }
```

It is true that a user has many articles; likewise, it is true that a particular article belongs to a particular user.

In the next chapter, you will see those relations in great detail.

CHAPTER 5

Database Migration and Eloquent

You can interact with databases using Laravel 5.8 in a simple and expressive way. Either you can use the DB facade, using raw SQL to query your databases, or you can use Eloquent ORM.

Laravel supports four databases.

- MySQL
- PostgreSQL
- SQLite
- SQL Server

You have learned how to configure a database for any application you want to build. The database configuration file is `config/database.php`. If you want, you can define all your database connections; you can also specify which one will be the default one.

Introduction to Migration

You can easily create and modify your database tables with the help of Laravel's schema builders. Migrations are typically paired with the schema builders. You can also easily share database tables with your team members.

How easy is it? Let's see an example first.

To create a migration, you use the command `make:migration create_tests_table`.

© Sanjib Sinha 2019
S. Sinha, *Beginning Laravel*, https://doi.org/10.1007/978-1-4842-4991-8_5

Suppose you are going to create a `tests` table. The command is simple, as shown here:

```
//code 5.1
php artisan make:migration create_tests_table
```

Every database table represents a particular resource. In this case, the `create_tests_table` table also connects to the `Test` model. You can create the same table at the same time as the model, or you can create table separately.

You saw earlier that a new migration was placed in the `database/migrations` directory, and each migration file contains a timestamp that helps Laravel to maintain the order of the migrations.

To return to the previous subject, there are two options that are used to indicate the name of the table and to indicate whether the migration is new. It is necessary to modify some functionality of any table. Another important aspect is that the name of the table always relates to the model name. If the model name is `Test`, for example, then Laravel guesses that the table name will be `create_tests_table`; Laravel adds a prefix of the timestamp before the table name, as in `2019_05_18_013613_create_tests_table`.

```
//code 5.2
php artisan make:migration create_tests_table -create=tests
php artisan make:migration add_names_to_tests_table -table=tests
```

Depending on the number of your resources, you can create as many tables as you need to create for your application. Later, through Eloquent ORM, you can build relationships between those tables. Laravel makes this extremely easy for developers.

Introduction to Eloquent

Working with a database is not always easy. Eloquent ORM provides a simple ActiveRecord implementation for working with a database. As I mentioned earlier, each database table has a corresponding model that represents a particular resource.

These models are used to interact with the corresponding database table. Each model allows you to query the data in the database table and insert, update, and delete data.

To start with, you can create any Eloquent model by issuing a simple command: `-make:model`. You will find that model in the `app` directory. All Eloquent models extends the `Illuminate\Database\Eloquent\Model` class. Through the `artisan` commands, you can create models in this way:

```
//code 5.3
php artisan make:model Test
```

To generate a database migration, you can add either the `--migration` or `--m` option, as shown here:

```
//code 5.4
php artisan make:model Test --migration
php artisan make:model Test -m
```

Once your model is created, you can start building the relationships between the database tables.

Introduction to Eloquent Relations

A good application rests on well-defined relationships between database tables. Considering each table and model as separate resources, you can define how they will interact with each other.

Suppose an article post has many comments or it has been associated with a particular username. In that case, you can say each article has *one-to-one* relation with the user. This relation is defined in the models `Article` and `User`. This is the same for comments also. Eloquent helps manage and work with these relationships, and it supports several different types of relationships. You will see them in a minute.

In the Eloquent model classes, the relationships are defined as methods. Actually, based on these relationships, you can make your queries. Therefore, these methods provide powerful method chaining and querying capabilities.

How can you chain your methods? Here is an example of adding additional constraints:

```
$user->articles()->where('category_id', 1)->get();
```

You'll now learn how to define each type separately. Let's start with one-to-one relationships. Then you will examine the others.

One-to-One

A one-to-one relationship is typically related to the basic algebraic function. One input passes through a function and gives one output, and they have one-to-one relation. At the same time, the inverse is also true.

This concept is old and hugely popular among scientists, as it defines one key root of mathematical operations.

Now, in our case, an article can belong to one user. In that scenario, the relationship I am talking about is one-to-one, and the inverse is also true. One user can write at least one article. But at the same time, one user can write many articles. So, the statement "many articles belong to one user" is also true. You will see those concepts in a minute.

Recall from Chapter 4 that the `Article` model class looks like this:

```
//code 5.5
//app/Article.php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Article extends Model
{
    //
    protected $fillable = [
        'user_id', 'title', 'body',
    ];

    public function user() {
        return $this->belongsTo('App\User');
    }

    /**
     * Get the tags for the article
    */

   public function tags() {
       return $this->belongsToMany('App\Tag');
   }
```

```
    /**
    * Get all of the profiles' comments.
    */
    public function comments(){
      return $this->morphMany('App\Comment', 'commentable');
    }
}
```

In the `Article` model class, you have probably noticed that relationships have been defined not only as methods but also as powerful query builders. When you define relationships, the methods provide powerful method chaining and querying capabilities.

```
{{ $article->user->name }}
```

So, from the `Article` model, you get the respective username of the author of that particular article. In the `app/User.php` file, you'll see this line of code:

```
public function article() {
        return $this->hasOne('App\Article');
    }
```

The first argument passed to the `hasOne('App\Article')` method is the name of the related model. You have seen this line of code in `app/Article.php`:

```
public function user() {
        return $this->belongsTo('App\User');
    }
```

The relationship has been defined in both models, and once it has been defined, you can retrieve the affiliated record using Eloquent's dynamically defined properties. These dynamically defined properties in Eloquent are so powerful that accessing relationship methods is a cakewalk. It seems as if they were defined in the model. You have seen the following example before:

```
{{ $article->user->name }}
```

You couldn't have the username from the `Article` model if Eloquent had not determined the foreign key of the relationship based on the model name. In this case, the `Article` model mechanically assumes that it has a `user_id` foreign key. Here the parent is the User model, and it has a users table that has a custom $primaryKey column (id). So, user_id of the Article model matches the ID of the parent User model.

Based on this assumption, you can take your application's functionalities further. You can grab the first article written by the user using the Article model. Consider this line of code:

```
{{ $article->user->find($article->user_id)->article->title }}
```

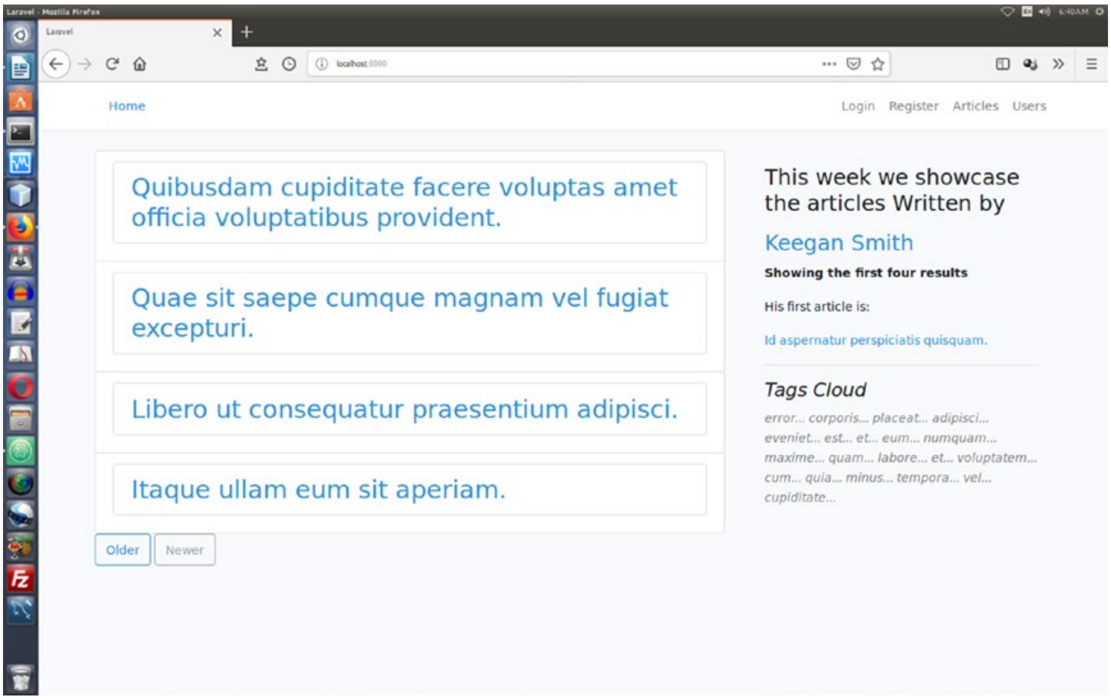
It is same as User::find(1)->article->title. However, if you wanted to grab the user's first-ever written article this way, you would have to go back to the related Controller class and add this line of code there. The flexibility of Eloquent ORM allows you to get the same result using the Article model. So, let's first look at the welcome.blade.php code:

```
// code 5.7
// resources/views/welcome.blade.php
@extends('layouts.app')
@section('content')
<div class="container">
    <div class="row">
        <div class="col-md-8 blog-main col-lg-8 blog-main col-sm-8 blog-
        main">
          <div class="blog-post">
    <ul class="list-group">
         @foreach($articles as $article)
         <li class="list-group-item">
             <h2 class="blog-post-title">
                 <li class="list-group-item"><a href="/articles/{{
                 $article->id }}">{{ $article->title }}</a>
             </h2>
         </li>
         @endforeach
    </ul>
            </div>
          <nav class="blog-pagination">
            <a class="btn btn-outline-primary" href="#">Older</a>
            <a class="btn btn-outline-secondary disabled" href="#">Newer</a>
          </nav>
        </div>
        <aside class="col-md-4 blog-sidebar">
          <div class="p-3">
              <h3 class="blog-post-title">This week you showcase the
              articles
                  Written by <p></p>
                  <a href="/users/{{ $article->user_id }}/articles">
                  {{ $article->user->name }}</a>
        </h3>
              <strong>Showing the first four results</strong><p></p>
              <italic>His first article is:</italic>
              <p></p>
              <a href="/articles/{{ $article->user->article->id }}">
                  {{ $article->user->find($article->user_id)->article->
                  title }}</a>
              <hr class="linenums" color="red">
              <h4 class="font-italic">Tags Cloud</h4>
            @foreach($tags as $tag)
            <font class="font-italic" color="gray"> {{ $tag->tag }}...
            @endforeach
          </div>
        </aside>
    </div>
</div>
@endsection
```

In the lower part of the file `welcome.blade.php`, you should change a segment to accommodate a one-to-one relation between the `article` and `user`, as shown here:

```
// code 5.7
// resources/views/welcome.blade.php
<aside class="col-md-4 blog-sidebar">
          <div class="p-3">
              <h3 class="blog-post-title">This week we showcase the articles
                  Written by <p></p>
          <a href="/users/{{ $article->user_id }}/articles">
          {{ $article->user->name }}</a>
</h3>
      <strong>Showing the first four results</strong><p></p>
      <italic>His first article is:</italic>
      <p></p>
      <a href="/articles/{{ $article->user->article->id }}">
          {{ $article->user->find($article->user_id)->article-
          >title }}</a>
      <hr class="linenums" color="red">
      <h4 class="font-italic">Tags Cloud</h4>
    @foreach($tags as $tag)
    <font class="font-italic" color="gray"> {{ $tag->tag }}...
    @endforeach
  </div>
</aside>
```

Now the front page looks like Figure 5-1.


Laravel - Mozilla Firefox

Laravel

localhost:8000

Home

Login Register Articles Users

Quibusdam cupiditate facere voluptas amet officia voluptatibus provident.

Quae sit saepe cumque magnam vel fugiat excepturi.

Libero ut consequatur praesentium adipisci.

Itaque ullam eum sit aperiam.

Older Newer

This week we showcase the articles Written by

Keegan Smith

Showing the first four results

His first article is:

Id aspernatur perspiciatis quisquam.

Tags Cloud

error... corporis... placeat... adipisci... eveniet... est... et... eum... numquam... maxime... quam... labore... et... voluptatem... cum... quia... minus... tempora... vel... cupiditate...

***Figure 5-1.** Home page of the content management application*

Let’s move on to the one-to-many relationship.

One-to-Many

Remember, a user has many articles, and an article has many tags. A user also has many tags. A user has many comments, and these relations between them go on and on. These all indicate a type of Eloquent relationship: one-to-many.

On the home page of the application, you have a link to all the articles written by the user. In the top-left sidebar of the home page, you can click the user’s name and read all the articles and a short profile also.

The link on the home page looks like this:

```
<a href="/users/{{ $article->user_id }}/articles">{{ $article->user->name
}}</a>
```

The URI is very expressive; it reads like this: `users/1/articles`. This means you want to read all the articles by the user with an ID of 1. This is an ideal candidate for a one-to-many relationship. On this page, you can also show the user’s profile.

Let’s work on registering the web routes. Here is the `routes/web.php` code as a whole:

```
//code 5.8
// routes/web.php
Auth::routes();
Route::get('/home', 'HomeController@index')->name('home');
Route::resources([
  'users' => 'UserController',
  'profiles' => 'ProfileController',
  'articles' => 'ArticleController',
  'comments' => 'CommentController'
]);
Route::get('/users/{id}/articles', 'ArticleController@articles');
Route::get('/', 'ArticleController@main');
```

In `routes/web.php`, you define the relationship in this way: `Route::get('/users/{id}/articles', 'ArticleController@articles')`. This means you should create the `articles()` method in the `ArticleController`. Let’s look at the code of

ArticleController as a whole first; after that, you will take a look at the articles method individually:

```
// code 5.9
// app/HTTP/Controllers/ArticleController.php
<?php

namespace App\Http\Controllers;

use App\Article;
use App\Country;
use App\User;
use App\Tag;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;

class ArticleController extends Controller
{
    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        $articles = Article::orderBy('created_at','desc')->paginate(4);
        //$articles = Article::where('active', 1)->orderBy('title',
        'desc')->take(10)->get();
        $users = User::all();
        $tags = Tag::all();
        //$posts = Article::orderBy('created_at','desc')->paginate(3);
        return view('articles.index', compact('articles', 'users',
        'tags'));
    }
```

```
public function main()
{
    $articles = Article::where('user_id', 1)->orderBy('title', 'desc')-
    >take(4)->get();
    $tags = Tag::all();
    return view('welcome', ['articles' => $articles, 'tags' => $tags]);
}

/**
 * Show the form for creating a new resource.
 *
 * @return \Illuminate\Http\Response
 */
public function create()
{
    //
}

/**
 * Store a newly created resource in storage.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return \Illuminate\Http\Response
 */
public function store(Request $request)
{
    //
}

/**
 * Display the specified resource.
 *
 * @param  \App\Article  $article
 * @return \Illuminate\Http\Response
 */
```

```
    public function show(Article $article)
    {
        $tags = Article::find($article->id)->tags;
        $article = Article::find($article->id);
        $comments = $article->comments;
        $user = User::find($article->user_id);
        $country = Country::where('id', $user->country_id)->get()->first();

        return view('articles.show', compact('tags','article',
        'country', 'comments', 'user'));
    }
     /**
     * Display the specified resource.
     *
     * @param  \App\Article  $article
     * @return \Illuminate\Http\Response
     */
    public function articles($id)
    {
        $user = User::find($id);

        return view('articles.articles', compact('user'));
    }

    /**
     * Show the form for editing the specified resource.
     *
     * @param  \App\Article  $article
     * @return \Illuminate\Http\Response
     */
    public function edit(Article $article)
    {
        //
    }
    /**
     * Update the specified resource in storage.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \App\Article  $article
     * @return \Illuminate\Http\Response
     */
    public function update(Request $request, Article $article)
    {
        //
    }

    /**
     * Remove the specified resource from storage.
     *
     * @param  \App\Article  $article
     * @return \Illuminate\Http\Response
     */
    public function destroy(Article $article)
    {
        //
    }
}
```

Now, you can take a look at the `articles` method to understand how one-to-many relations work here. Here is the code:

```
public function articles($id)
    {
        $user = User::find($id);
        return view('articles.articles', compact('user'));
    }
```

Here you pass the user ID, and using the Eloquent method `find(id)`, you get the respective user. After that, you pass the `user` object to the `resource/views/articles/articles.blade.php` file. Next, you should look at both the `User` and `Article` models to recall how to define the one-to-many relationship. A user has many articles.

So, you wrote this in Chapter 4:

```
// code 5.10
// app.User.php
public function articles() {
        return $this->hasMany('App\Article');
    }
```

At the same time, you need to look at the inverse model relationship in the `Article` model. Say you want to access all of a user's articles. You have already defined a relationship to allow an article to access its parent user as part of the one-to-one relationship. This is enough to define the inverse of a `hasMany` relationship, as shown here:

```
// code 5.11
// app/Article.php
public function user() {
        return $this->belongsTo('App\User');
    }
```

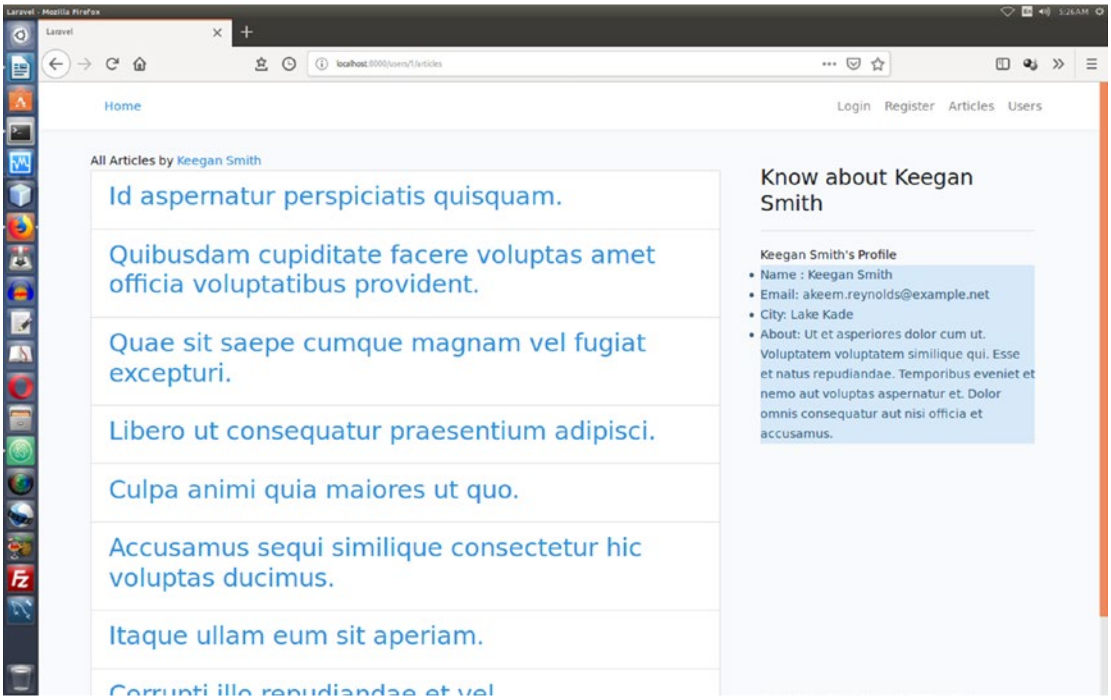
Let's see how you can access all the articles written by one particular user. In the top-left sidebar of the home page of your application, you have the link that takes users to the desired page. I'm not going to show every line of `articles.blade.php` anymore. For brevity, you will look at the vital portions, starting with this:

```
// code 5.14
// resources/views/articles.blade.php
<ul class="list-group">
         <div class="panel-heading">All Articles by <a href="/users/{{
         $user->id }}">{{ $user->name }}</a> </div>
                @foreach($user->articles as $article)
                <li class="list-group-item">
                    <h2 class="blog-post-title">
                        <a href="/articles/{{ $article->id }}">{{ $article->
                        title }}</a>
                    </h2>
                </li>
```

```
            @endforeach
    </ul>
...
<div class="p-3">
              <h3 class="blog-post-title">Know about {{ $article->user-
              >name }}
              </h3>
              <hr class="linenums" color="red">
              <div class="panel panel-default">
                <div class="panel-heading">{{ $article->user->name }}'s
                Profile</div>
                <div class="panel-body">
                    <li class="list-group-item-info">Name : {{ $article-
                    >user->name }}</li>
                    <li class="list-group-item-info">Email: {{ $article->
                    user->email }}</li>
                    <li class="list-group-item-info">City: {{ $article->
                    user->profile->city }}</li>
                    <li class="list-group-item-info">About: {{ $article->
                    user->profile->about }}</li>
                </div>
            </div>
          </div>
```

Look at this line: `@foreach($user->articles as $article)`. Since the `User` model has a one-to-many relationship with the `Article` model, you can access all the articles by any particular user quite easily.

As you have probably noticed, I have chained conditions onto the query. Since the relationship has been defined, you can access the collection of articles by accessing the `articles` property. This is the magic of Eloquent: it provides "dynamic properties." Therefore, you can access relationship methods as if they were defined as properties on the model. Go to `http://localhost:8000/users/1/articles` in your browser (see Figure 5-2).



All Articles by Keegan Smith

Id aspernatur perspiciatis quisquam.

Quibusdam cupiditate facere voluptas amet officia voluptatibus provident.

Quae sit saepe cumque magnam vel fugiat excepturi.

Libero ut consequatur praesentium adipisci.

Culpa animi quia maiores ut quo.

Accusamus sequi similique consectetur hic voluptas ducimus.

Itaque ullam eum sit aperiam.

Corrupti illo repudiandae et vel

Know about Keegan Smith

Keegan Smith's Profile

- Name : Keegan Smith
- Email: akeem.reynolds@example.net
- City: Lake Kade
- About: Ut et asperiores dolor cum ut. Voluptatem voluptatem similique qui. Esse et natus repudiandae. Temporibus eveniet et nemo aut voluptas aspernatur et. Dolor omnis consequatur aut nisi officia et accusamus.

Figure 5-2. *All articles by a single user along with the profile*

The one-to-one relationship between `User` and `Profile` allows you to get the profile data of any particular user as well (Figure 5-2). The magic of Eloquent dynamic properties is happening splendidly. You define them in the models and later access them in the controller. The controller transports that data to the views.

Before explaining many-to-many relationship, I'll discuss one more important concept: dependency injection and separation of concerns.

Separation of Concerns

The previous pieces of code in the controllers are concise, but you are unable to test them without hitting the database directly. Your Eloquent ORM is tightly coupled with your controller. This is not desirable because this is violating the design principle known as *separation of concerns* that demands that every class should have a single responsibility and that should be encapsulated by the class itself. In other words, your controller should not know the data source. In this case, your web layer and the data access layer are tightly coupled. To decouple them, you can inject a user repository class that can be coupled with Eloquent ORM, making your `Controller` class free.

To do that, you can add one `Repositories` folder in app. Inside the `Repositories` folder, you can add two more folders: `DBRepositories` and `Interfaces`. Inside the `Interfaces` folder, you can add an interface called `UserRepositoryInterface`.

```
//code 5.15
//app/Repositories/ Interfaces/UserRepositoryInterface.php
<?php namespace RepositoryInterface;

 interface UserRepositoryInterface {
     public function all();
}
```

Now, inside the `DBRepositories` folder, you can add a database repository class.

```
// code 5.15
//app/ Repositories/DBRepositories/DBUserRepository.php
<?php namespace RepositoryDB;

use RepositoryInterface\UserRepositoryInterface as UserRepositoryInterface;

use App\User;
use Illuminate\Http\Request;

class DBUserRepository implements UserRepositoryInterface {
    public function all() {
        return User::all();
    }
}
```

Now, you want to inject this database repositories class into your controller. However, to make it work, you need to update your `composer.json` file a little bit.

```
//code 5.16
//composer.json
"autoload": {
        "classmap": [
            "database/seeds",
            "database/factories"
        ],
        "psr-4": {
            "App\\": "app/",
            "RepositoryInterface\\": "app/Repositories/Interfaces/",
            "RepositoryDB\\": "app/Repositories/DBRepositories/"
        }
    },
```

You need to run this command on your terminal first, before modifying the User controller.

```
// code 5.17
$ composer dump-autoload
Cannot create cache directory /home/ss/.composer/cache/repo/https---
packagist.org/, or directory is not writable. Proceeding without cache
Cannot create cache directory /home/ss/.composer/cache/files/, or directory
is not writable. Proceeding without cache
Generating optimized autoload files
> Illuminate\Foundation\ComposerScripts::postAutoloadDump
> @php artisan package:discover
Discovered Package: beyondcode/laravel-dump-server
Discovered Package: fideloper/proxy
Discovered Package: laravel/tinker
Discovered Package: nesbot/carbon
Discovered Package: nunomaduro/collision
Package manifest generated successfully.
```

Now you can modify your UserController in this way:

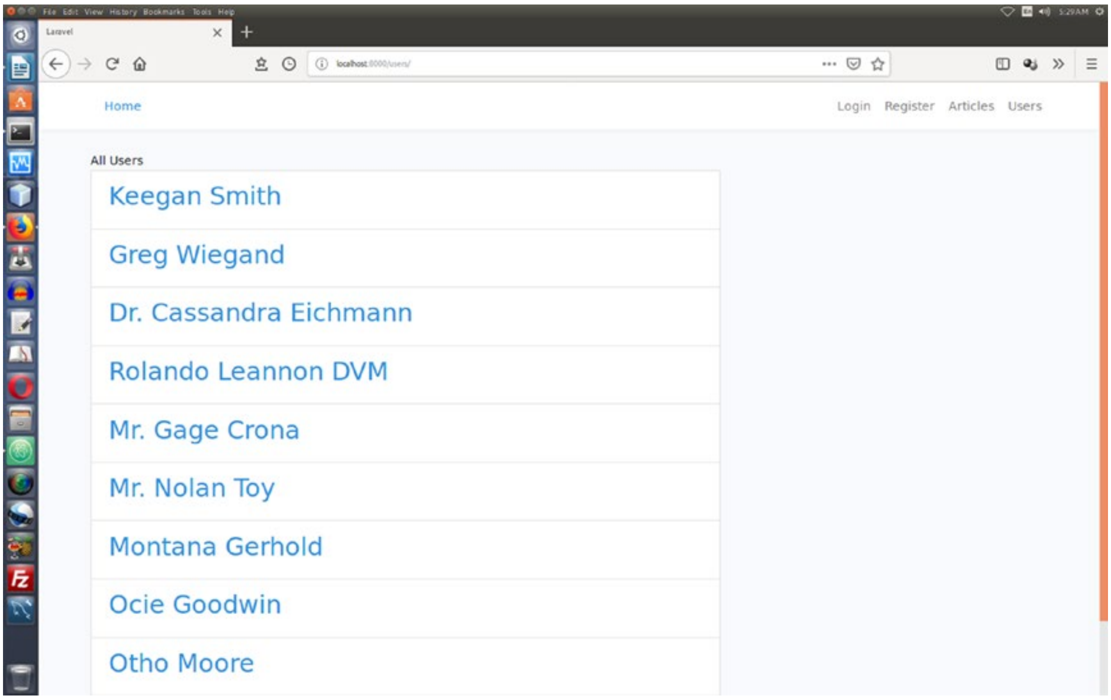
```
// code 5.18
//app/HTTP/Controllers/UserController.php
<?php

namespace App\Http\Controllers;

use App\User;
use Illuminate\Http\Request;

use RepositoryDB\DBUserRepository as DBUserRepository;
//use RepositoryInterface\UserRepositoryInterface as UserRepositoryInterface;
```

```
class UserController extends Controller {
    public $users;
    public function __construct(DBUserRepository $users) {
        $this->users = $users;
    }

    public function index(){
        $users = $this->users->all();
        return view('users.index', compact('users'));
    }
}
```

Your controller does not know the database source. It is completely ignorant of the data access layer. It only transports the database repositories objects to the view. The `resources/views/users/index.blade.php` code will look like this:

```
//code 5.19
//resources/views/users/index.blade.php
@foreach($users as $user)
                <li class="list-group-item">
                    <h2 class="blockquote-reverse">
                        <a href="/users/{{ $user->id }}">{{ $user->name }}</a>
                    </h2>
                </li>
                @endforeach
```

Now type `http://localhost:8000/users` in your browser, and you will get the output in Figure 5-3.



All Users

Keegan Smith
Greg Wiegand
Dr. Cassandra Eichmann
Rolando Leannon DVM
Mr. Gage Crona
Mr. Nolan Toy
Montana Gerhold
Ocie Goodwin
Otho Moore

Figure 5-3. *Output of all users*

This dependency injection is good for testability. At the same time, it gives you ample chance to decouple your classes and keep the principle of separation of concerns intact.

Many-to-Many

An example of many-to-many relations here is articles and tags. It is slightly more complicated than the one-to-one and one-to-many relations. When the many-to-many relationship exists between articles and tags, the tags are shared by other articles. Many articles may have one particular tag.

You have already created three tables: `articles`, `tags`, and `article_tag`.

Tip There is one thing to remember here. Whenever you create a pivot table (here `article_tag`), the naming must be in alphabetical order of the related model names. Alphabetically, `Article` comes before `Tag`. So, you name this table as `article_tag`. This table must contain the `article_id` and `role_id` columns.

I will discuss `Tag` and `Comment` later in detail; before that, I'd like to point out one key concept carefully. It is obvious, from the relationships between models, that an article has many tags, or vice versa. In plain words, this is perfectly logical. An article has many tags indeed. Inversely, a tag may belong to many articles too. Keeping this logic in mind, you can add the following code to your `Article` model:

```
// code 5.12
// app/Article.php
/**
     * Get the tags for the article
    */
    public function tags(){
        return $this->hasMany('App\Tag');
    }
```

Now you should get all tags that the article has. You can use this line of code in the `show()` method of `ArticleController.php`:

```
public function show(Article $article)
    {
        $tags = Article::find($article->id)->tags;
        $article = Article::find($article->id);
        return view('articles.show', compact('article', 'tags'));
    }
```

Although this looks perfectly sane, it will produce errors like this:

> "SQLSTATE[42S22]: Column not found: 1054 Unknown column 'tags.article_id' in 'where clause' (SQL: select * from `tags` where `tags`.`article_id` = 14 and `tags`.`article_id` is not null) ◀"

Why did this happen? You used the same properties before in the `User` model. There you accessed user ➤ articles quite easily without any trouble. Then why can't you get articles ➤ tags in the same manner?

Here, you need to understand one key concept. In this case, the one-to-many relationship should allow Eloquent to assume that the foreign key in the `Tag` model is `article_id`. That you don't have in the `tags` table. Instead, you have a pivot table called article_tag where you have two foreign keys: article_id and tag_id. So, if you wrote it like this:

```
// code 5.13
// app/Article.php
class Article extends Model
    {
        /**
         * Get the tags for the article.
         */
        public function tags()
        {
            return $this->hasMany('App\Tag');
        }
    }
```

it would throw an error. And that's exactly what happens here.

Eloquent has the ability to determine the proper foreign key column on the Tag model. It is a convention that Eloquent will take the snake_case name of the parent model (here Article) and add the suffix _id. This means, in this case, Eloquent will assume the foreign key on the Tag model is article_id. In this case, Eloquent didn't find the column tags.article_id and says that it is an unknown column.

Therefore, in the case of a one-to-many relationship, a foreign key should explicitly be defined. Before migration, when you created the articles table, you added this line:

```
$table->integer('user_id');
```

So, be careful about using one-to-many relationships and keep this in mind. Here the relationship between articles and tags is many-to-many. And for that you have a pivot table between them. I will discuss this in detail in the next section.

Now you should define the many-to-many relationship by writing a method that returns the result of the belongsToMany method. First, you define the tags method on your Article model, as shown here:

```
// code 5.20
// app/Article.php
/**
     * Get the tags for the article
    */
    public function tags() {
        return $this->belongsToMany('App\Tag');
    }
```

Once the relationship is defined, you can access the article's tags through the dynamic property of `tags`. In `resources/views/articles/show.blade.php`, you have accessed the article's tags in this way:

```
@foreach($article->tags as $tag)
                    {{ $tag->tag }} ,
 @endforeach
```

The same is true for the `role_user` table. Here, `role` comes before the user. And in the respective `Role` and `User` models, you have already defined the many-to-many relationships by using the `belongsTo` and `belongsToMany` methods. Now, you want three types of users: `Administrator`, `Moderator`, and `Member`. Since you have used `Faker` to create three arbitrary words for the `roles` table, you need to change the name manually. Another important thing is you don't need too many administrators. So in the `database/seeds/DatabaseSeeder.php` file, I have limited the number of rows to three by using this code:

```
factory(App\Role::class, 3)->create()->each(function($role){
          $ids = range(1, 2);
          shuffle($ids);
          $sliced = array_slice($ids, 1, 5);
          $role->users()->attach($sliced);
        });
```

In the next step, you change the `UserController.php show()` method to this:

```
public function show(User $user)
    {
      $roles = User::find($user->id)->roles;
        $user = User::find($user->id);
        return view('users.show', compact('user', 'roles'));
    }
```

Now, except for user 1, other users have no role at present. But user 1 has been assigned to three roles by the `Faker` object. But for other users it would be different. Since you have assigned a role to user 1, the other users have no roles to show up. These users belong to the general member category.

I am trying to keep things simple so that you can understand the model relations. You want users to have some types of restrictions on their movements. Every user can post, but unless the administrator gives permission, the post will not be published. The same is true for the comments. You want your moderators to have some responsibilities, such as the ability to edit any post.

Everything depends on the model relations you define in your models and respective tables. By using them judiciously, you can build an awesome content management application.

Has-Many-Through

A distant relationship looks simple, but in reality, when you want to create one via an intermediary table, it can be complicated.

In the example content management application, you have already seen many interesting relationships. However, you have not found any distant relationship yet. For an example, let's suppose you have a table called `countries`. It has only an `Id` and `Name`. Now let's add an extra column in the `users` table; suppose it is `country_id`. As the `articles` table has already been assigned `user_id`, it is possible to establish a distant relationship between the `Country` and `Article` models (or, in other words, between the `countries` and `articles` tables). A particular user belongs to a country; because of that, you can grab all articles written by the users belonging to the same country.

Let's continue developing the application to see how this has-many-through relation works.

First create a `Country` model and a `countries` table using one command, as shown here:

```
$ php artisan make:model Country -m
```

Let's first define the `Country` model.

```
// code 5.21
// app/Country.php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
```

```
class Country extends Model
{

  protected $fillable = [
      'name'
  ];

  public function users() {
      return $this->hasMany('App\User');
  }

  public function articles(){
      return $this->hasManyThrough('App\Article', 'App\User');
  }
}
```

Let me explain this code. A `Country` model will probably have many users. The second method, `articles`, returns a special has-many-through relationship between two models, `Article` and `User`, as I have mentioned. Because of this method, you can access all the articles for any given country.

In the `countries` table, you must have this method ready before you populate the table with `Faker`.

```
// code 5.22
// database/migrations/countries table
public function up()
    {
        Schema::create('countries', function (Blueprint $table) {
            $table->increments('id');
            $table->string('name');
            $table->timestamps();
        });
    }
```

You also need to update your `users` table like this:

```
// database/migrations/users table
public function up()
    {
        Schema::create('users', function (Blueprint $table) {
            $table->increments('id');
            $table->integer('country_id');
            $table->string('name');
            $table->string('email')->unique();
            $table->timestamp('email_verified_at')->nullable();
            $table->string('password');
            $table->rememberToken();
            $table->timestamps();
        });
    }
```

You have to update your `User` model with this method:

```
public function country() {
        return $this->belongsTo('App\Country');
    }
```

Next, you need to update your `UserFactory` code in this way:

```
// database/factories/UserFactory.php
$factory->define(App\User::class, function (Faker $faker) {
    return [
      'country_id' => $faker->randomDigit,
        'name' => $faker->name,
        'email' => $faker->unique()->safeEmail,
        'password' => '$2y$10$TKh8H1.PfQx37YgCzwiKb.
        KjNyWgaHb9cbcoQgdIVFlYg7B77UdFm', // secret
        'remember_token' => str_random(10),
    ];
});
```

Finally, you need to add this line to the database seeder file.

```
// database/seeds/DatabaseSeeder.php
factory(App\Country::class, 20)->create();
```

Now you can run this command to populate your tables with new Faker data:

```
$ php artisan migrate:refresh -seed
```

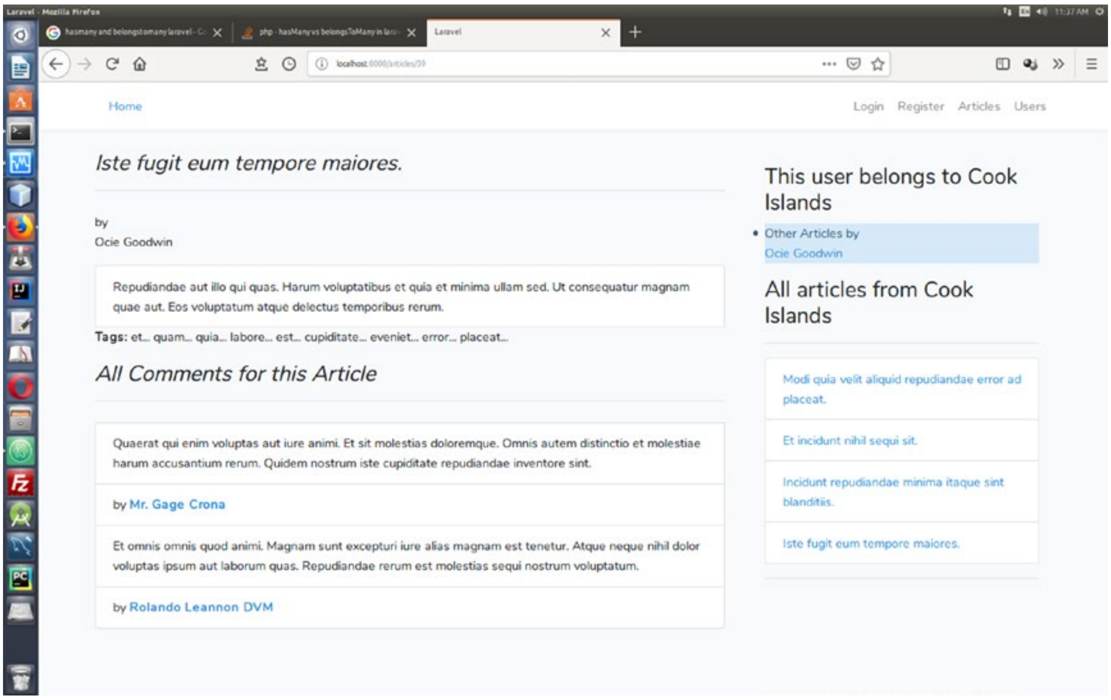
To see the effect, you must open ArticleController and update the show() method this way:

```
// code 5.51
//app/HTTP/Controllers/ ArticleController.php
public function show(Article $article)
    {
        $tags = Article::find($article->id)->tags;
        $article = Article::find($article->id);
        $user = User::find($article->user_id);
        $country = Country::where('id', $user->country_id)->get()->first();

        return view('articles.show', compact('article', 'tags',
        'country'));
    }
```

You have sent three objects, article, tags, and country, to your views. So, let's see the updated code of resources/views/articles/show.blade.php:

```
//code 5.23
//resources/views/articles/show.blade.php
@extends('layouts.app')

@section('content')
<div class="container">
    <div class="row">
        <div class="col-md-12 col-md-offset-2">
            <div class="panel panel-default">
                <div class="panel-heading">
                    <h3 class="pb-3 mb-4 font-italic border-bottom">{{
                    $article->title }}</h3>  by
                    <p>{{ $article->user->name }}</p>
                </div>
                <div class="panel-body">
                    <li class="list-group-item">{{ $article->body }}</li>
                    Tags:
                    @foreach($tags as $tag)
                    {{ $tag->tag }}...
                    @endforeach
                    <li class="list-group-item-info">Other Articles by
                        <p>
                            <a href="/users/{{ $article->user_id }}/
                            articles">{{ $article->user->name }}</a>
                        </p>
                        This user belongs to {{ $country->name }}<p></p>
                    </li>

                    <h3 class="blog-post">
                      All articles from {{ $country->name }}
                    </h3>
                    @foreach($country->articles as $article)
                    <li class="list-group-item">
                      <a href="/articles/{{ $article->id }}">
                        {{ $article->title }}
                      </a>
                    </li>
                    @endforeach
                </div>
            </div>
        </div>
    </div>
</div>
@endsection
```

Now if you click any article from the main articles page, you may have a view like Figure 5-4 where you get all articles for any given country.



Iste fugit eum tempore maiores.

by
Ocie Goodwin

Repudiandae aut illo qui quas. Harum voluptatibus et quia et minima ullam sed. Ut consequatur magnam quae aut. Eos voluptatum atque delectus temporibus rerum.

Tags: et... quam... quia... labore... est... cupiditate... eveniet... error... placeat...

All Comments for this Article

Quaerat qui enim voluptas aut iure animi. Et sit molestias doloremque. Omnis autem distinctio et molestiae harum accusantium rerum. Quidem nostrum iste cupiditate repudiandae inventore sint.

by Mr. Gage Crona

Et omnis omnis quod animi. Magnam sunt excepturi iure alias magnam est tenetur. Atque neque nihil dolor voluptas ipsum aut laborum quas. Repudiandae rerum est molestias sequi nostrum voluptatum.

by Rolando Leannon DVM

This user belongs to Cook Islands

- Other Articles by Ocie Goodwin

All articles from Cook Islands

Modi quia velit aliquid repudiandae error ad placeat.

Et incidunt nihil sequi sit.

Incidunt repudiandae minima itaque sint blanditiis.

Iste fugit eum tempore maiores.

Figure 5-4. *Accessing all articles for a given country, here from Cook Islands*

How does this happen? What are the mechanisms behind it? Consider this code in the `Article` model:

```
/**
         * Get all of the posts for the country.
         */
        public function articles()
        {
            return $this->hasManyThrough('App\Article', 'App\User');
        }
```

As you have seen, you have accessed articles by the `Country` model and got all the articles for a given country. So, the first argument passed to the `hasManyThrough` method is the name of the final model (here, `Article`) you want to access. The second argument is the name of the intermediate model (here, `User`).

To run this mechanism without any problems, typical Eloquent foreign key conventions are used. Why are they needed? Basically, they perform the relationship's queries. You can customize the keys of the relationship. To do that, you can follow this convention where you may pass them as the third, fourth, fifth, and sixth arguments:

```
class Country extends Model
{
    public function articles()
    {
        return $this->hasManyThrough(
            'App\Article',
            'App\User',
            'country_id', // Foreign key on users table...
            'user_id', // Foreign key on articles table...
            'id', // Local key on countries table...
            'id' // Local key on users table...
        );
    }
}
```

This has-many-through relationship is one of the strongest features of Laravel. So, be adventurous and use it to make your application truly awesome.

In the next section, I will discuss another great feature of Laravel model relations: polymorphic relationships that handle more complex relations.

Polymorphic Relations

Think about a model that belongs to more than one other model on a single association. Suppose you have a model called `Comment`. The users of your application can comment on articles written by other users. This is not at all difficult to build. Now, you also want one user to be able to comment on the profile of more than one other user. You can always bridge the `Comment` model either with the `Article` model or with the `Profile` model. You have seen how one-to-one, one-to-many, and many-to-many relations work.

However, it is difficult to imagine users of your application commenting on both articles and profiles using a single `comment` table. Polymorphic relations can handle this; with them, you can use a single `comment` table for both cases.

Moreover, you don't need any assistance from a pivot table to make it happen. All you need are two tables like `articles` and `profiles`, which have no direct relations with the `comment` table. Articles may have three simple columns, such as `id`, `title`, and `body`; and the `profiles` table may have two or three simple columns such as `id`, `name`, and `location`. Here the most interesting part is played by the `comments` table. It has columns like this: `id`, `body`, `commentable_id`, and `commentable_type`.

The all-important columns to note are the last two: `commentable_id` and `commentable_type`. The `commentable_id` column will contain the ID value of either articles or profiles. On the other hand, the `commentable_type` column will contain the class name of the owning model. When you post a comment on an article, the `commentable_type` column will contain a value like `App\Article` that represents the owning model. Eloquent ORM decides which type of owning model to return while accessing the `commentable` relation.

You have had enough theory! Let's start coding and see how you can apply this polymorphic relationship in the content management application.

The Problem

Say you want your application to reflect this polymorphic relationship between the `articles`, `profiles`, and `comments` table. You should do this in such a way that each article page will show how many commentators have posted comments on that particular article. You also want to show the user's name who has posted that comment. The same rule applies to the `profiles` table also. Any user can come and post on any profile page.

Since you want each comment to have a username associated with it, let's add another column called `user_id`.

You should plan your database seeder class in a new way so that it will populate the database tables like you want. The other tables were settled before, but the `user_id` column of the `comments` table should have different user ID. You have another great challenge ahead: the `commentable_type` column of the `comments` table should shuffle the class name of the two owning models: `Article` and `Profile`. You need to design your database seeder file in that way. You'll see how in the next section.

The Solution

Create the Comment model and comments table by issuing a single command, as shown here:

```
$ php artisan make:model Comment -m
```

Let's first start with the comments table code:

```
//code 5.24
//database/migrations/comments table
class CreateCommentsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
     public function up()
     {
         Schema::create('comments', function (Blueprint $table) {
             $table->increments('id');
             $table->integer('user_id');
             $table->text('body');
             $table->integer('commentable_id');
             $table->string('commentable_type');
             $table->timestamps();
         });
     }

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

Next you will see the full code snippets of the models: `Comment`, `Article`, and `Profile`. Let's start with the `Comment` model code, and the other two will follow it.

```
// code 5.25
// app/Comment.php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model
{
  /**
  * Get all of the owning commentable models.
  */
  public function commentable(){
    return $this->morphTo();
  }

  public function user() {
      return $this->belongsTo('App\User');
  }

  public function users() {
      return $this->belongsToMany('App\User');
  }

  public function article() {
      return $this->belongsTo('App\Article');
  }
  public function articles() {
      return $this->belongsToMany('App\Article');
  }

}
```

Here is the code for the `Article` model:

```
// app/Article.php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Article extends Model
{
    //
    protected $fillable = [
        'user_id', 'title', 'body',
    ];

    public function user() {
        return $this->belongsTo('App\User');
    }

    /**
     * Get the tags for the article
    */

   public function tags() {
       return $this->belongsToMany('App\Tag');
   }

   /**
   * Get all of the profiles' comments.
   */
   public function comments(){
     return $this->morphMany('App\Comment', 'commentable');
   }
}
```

Next, you will see the code of the `Profile` model, with an added line about comments:

```
// app/Profile.php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Profile extends Model
{
    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = [
        'user_id', 'city', 'about',
    ];

    public function user() {

        return $this->belongsTo('App\User');

    }

    /**
    * Get all of the profiles' comments.
    */
    public function comments(){
      return $this->morphMany('App\Comment', 'commentable');
    }
}
```

You also need to update the `User` model a little bit. Here is the full code snippet:

```
// app/User.php
<?php

namespace App;
```

```
use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = [
        'name', 'email', 'password',
    ];

    /**
     * The attributes that should be hidden for arrays.
     *
     * @var array
     */
    protected $hidden = [
        'password', 'remember_token',
    ];

    public function profile() {

        return $this->hasOne('App\Profile');

    }

    public function article() {
        return $this->hasOne('App\Article');
    }

    public function articles() {
        return $this->hasMany('App\Article');
    }
    public function role() {
        return $this->hasOne('App\Role');
    }

    public function roles() {
        return $this->hasMany('App\Role');
    }

    public function country() {
        return $this->belongsTo('App\Country');
    }

    public function comment() {
        return $this->hasOne('App\Comment');
    }

    public function comments() {
        return $this->hasMany('App\Comment');
    }
}
```

Now you have the models in place, ready to interact with each other. Let's populate all the tables by designing a `User` factory and database seeder files properly.

First, here is the user factory's full code snippet:

```
//code 5.26
// database/factories/UserFactory.php
<?php

use Faker\Generator as Faker;

/*
|--------------------------------------------------------------------------
| Model Factories
|--------------------------------------------------------------------------
|
| This directory should contain each of the model factory definitions for
| your application. Factories provide a convenient way to generate new
| model instances for testing / seeding your application's database.
|
*/
```

```
$factory->define(App\User::class, function (Faker $faker) {
    return [
      'country_id' => $faker->biasedNumberBetween($min = 1, $max = 20,
      $function = 'sqrt'),
        'name' => $faker->name,
        'email' => $faker->unique()->safeEmail,
        'password' => '$2y$10$TKh8H1.PfQx37YgCzwiKb.
        KjNyWgaHb9cbcoQgdIVFlYg7B77UdFm', // secret
        'remember_token' => str_random(10),
    ];
});

$factory->define(App\Article::class, function (Faker $faker) {
    return [
        'user_id' => App\User::all()->random()->id,
        'title' => $faker->sentence,
        'body' => $faker->paragraph(random_int(3, 5))
    ];
});

$factory->define(App\Profile::class, function (Faker $faker) {
    return [
        'user_id' => App\User::all()->random()->id,
        'city' => $faker->city,
        'about' => $faker->paragraph(random_int(3, 5))
    ];
});

$factory->define(App\Tag::class, function (Faker $faker) {
    return [
        'tag' => $faker->word
    ];
});

$factory->define(App\Role::class, function (Faker $faker) {
    return [
        'name' => $faker->word
    ];
});

$factory->define(App\Country::class, function (Faker $faker) {
    return [
        'name' => $faker->country
    ];
});

$factory->define(App\Comment::class, function (Faker $faker) {

    return [
      'user_id' => $faker->biasedNumberBetween($min = 1, $max = 10,
      $function = 'sqrt'),
        'body' => $faker->paragraph(random_int(3, 5)),
        'commentable_id' => $faker->randomDigit,
        'commentable_type' => function(){
          $input = ['App\Article', 'App\Profile'];
          $model = $input[mt_rand(0, count($input) - 1)];
          return $model;
        }
    ];
});
```

Note It is strongly recommended that you visit the GitHub repositories of the PHP faker. This will give you an idea of how you can use different attributes or methods to populate your application with fake data.

In the previous code snippet (code 5.26), the last section is interesting, as you have dealt with the `Comment` model before. Let's see that part again and try to understand what you have actually done.

```
$factory->define(App\Comment::class, function (Faker $faker) {
    return [
      'user_id' => $faker->biasedNumberBetween($min = 1, $max = 10,
      $function = 'sqrt'),
        'body' => $faker->paragraph(random_int(3, 5)),
        'commentable_id' => $faker->randomDigit,
        'commentable_type' => function(){
          $input = ['App\Article', 'App\Profile'];
          $model = $input[mt_rand(0, count($input) - 1)];
          return $model;
        }
    ];
});
```

The first line deals with `user_id`, and you know that you have ten users so far. You could have made it bigger, but that would not serve the purpose here of understanding model relations. You use a special faker method that will check for a range (1 to 10) and populate the table accordingly.

Populating the `commentable_type` column has been a real challenge as you want to populate it with both the `App\Article` and `App\Profile` model class names randomly. So, you use an anonymous function that returns an array and shuffles out two values.

Next, you will see the database seeder file. Although you are updating only the `comment` part, you should take a look at the full code snippet, shown here:

```
//code 5.27
//database/seeds/DatabaseSeeder.php
<?php

use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     *
     * @return void
     */
    public function run()
    {
        // $this->call(UsersTableSeeder::class);
        // $this->call(UsersTableSeeder::class);
        factory(App\User::class, 10)->create()->each(function($user){
            $user->profile()->save(factory(App\Profile::class)->make());
        });

        factory(App\Tag::class, 20)->create();

        factory(App\Country::class, 20)->create();

        factory(App\Comment::class, 50)->create();

        factory(App\Article::class, 50)->create()->each(function($article){
          $ids = range(1, 50);
          shuffle($ids);
          $sliced = array_slice($ids, 1, 20);
          $article->tags()->attach($sliced);
        });

        factory(App\Role::class, 3)->create()->each(function($role){
          $ids = range(1, 2);
          shuffle($ids);
          $sliced = array_slice($ids, 1, 5);
          $role->users()->attach($sliced);
        });
    }
}
```

Now you are ready to refresh the seed data with the following command that will populate all the tables:

```
$ php artisan migrate:refresh -seed
```

You have successfully seeded your tables with fake data, so now you are also ready to test your applications. Before moving on to that final part, let's see how your database tables look. First, let's open your `phpMyAdmin` interface and select the database you have used to test this application (Figure 5-5).

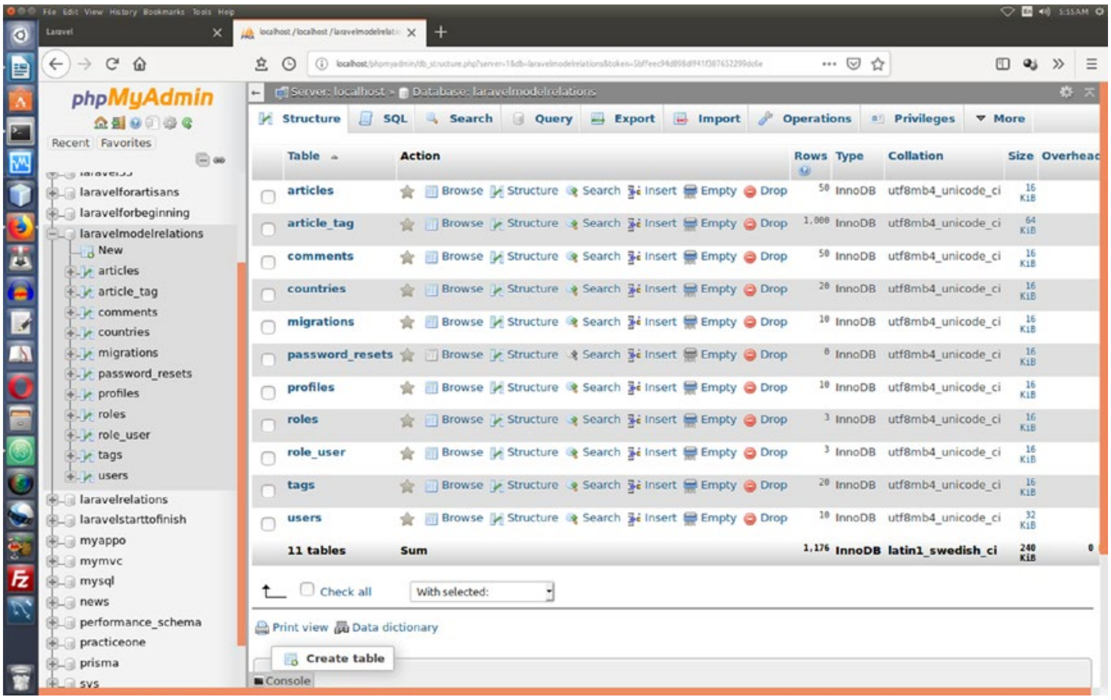
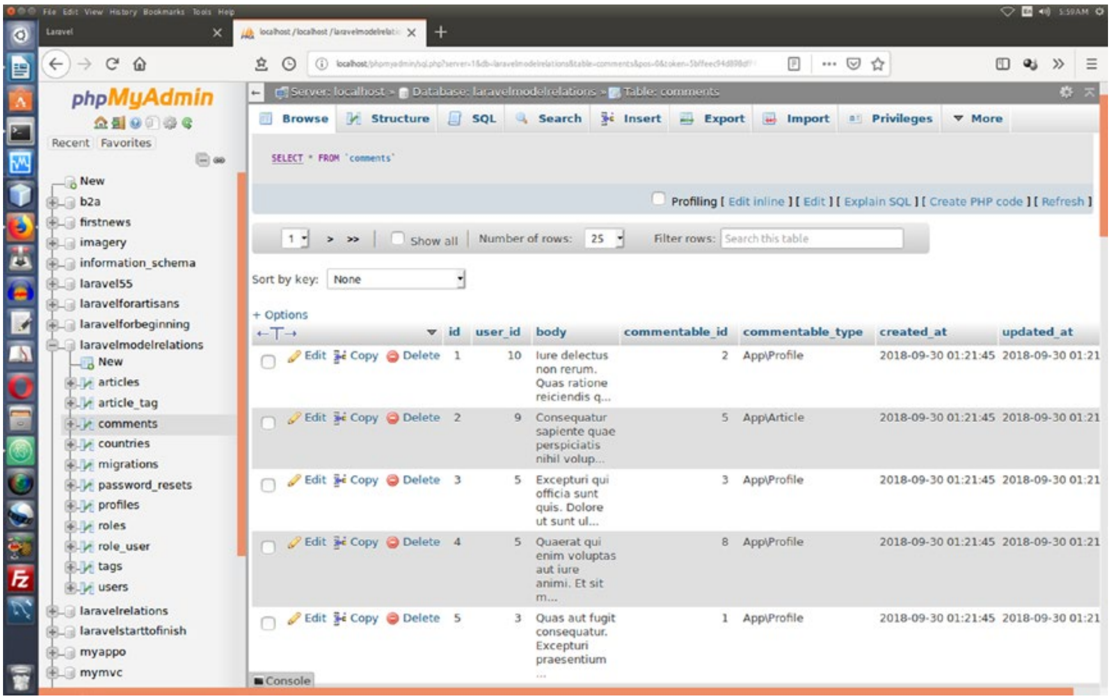


Table	Action	Rows	Type	Collation	Size	Overhead
articles	Browse Structure Search Insert Empty Drop	50	InnoDB	utf8mb4_unicode_ci	16 KiB	
article_tag	Browse Structure Search Insert Empty Drop	1,000	InnoDB	utf8mb4_unicode_ci	64 KiB	
comments	Browse Structure Search Insert Empty Drop	50	InnoDB	utf8mb4_unicode_ci	16 KiB	
countries	Browse Structure Search Insert Empty Drop	20	InnoDB	utf8mb4_unicode_ci	16 KiB	
migrations	Browse Structure Search Insert Empty Drop	10	InnoDB	utf8mb4_unicode_ci	16 KiB	
password_resets	Browse Structure Search Insert Empty Drop	0	InnoDB	utf8mb4_unicode_ci	16 KiB	
profiles	Browse Structure Search Insert Empty Drop	10	InnoDB	utf8mb4_unicode_ci	16 KiB	
roles	Browse Structure Search Insert Empty Drop	3	InnoDB	utf8mb4_unicode_ci	16 KiB	
role_user	Browse Structure Search Insert Empty Drop	3	InnoDB	utf8mb4_unicode_ci	16 KiB	
tags	Browse Structure Search Insert Empty Drop	20	InnoDB	utf8mb4_unicode_ci	16 KiB	
users	Browse Structure Search Insert Empty Drop	10	InnoDB	utf8mb4_unicode_ci	32 KiB	
11 tables	Sum	1,176	InnoDB	latin1_swedish_ci	240 KiB	0

***Figure 5-5.** Your full database in PHPMyAdmin*

Next, let's see how your `comments` table looks in `phpMyAdmin`. You will also look at it in a terminal using the MySQL prompt. But before that, you can take a look at `phpMyAdmin`, as shown in Figure 5-6.



```
SELECT * FROM `comments`
```

id	user_id	body	commentable_id	commentable_type	created_at	updated_at
1	10	Iure delectus non rerum. Quas ratione reiciendis q...	2	App\Profile	2018-09-30 01:21:45	2018-09-30 01:21
2	9	Consequatur sapiente quae perspiciatis nihil volup...	5	App\Article	2018-09-30 01:21:45	2018-09-30 01:21
3	5	Excepturi qui officia sunt quis. Dolore ut sunt ul...	3	App\Profile	2018-09-30 01:21:45	2018-09-30 01:21
4	5	Quaerat qui enim voluptas aut iure animi. Et sit m...	8	App\Profile	2018-09-30 01:21:45	2018-09-30 01:21
5	3	Quas aut fugit consequatur. Excepturi praesentium ...	1	App\Profile	2018-09-30 01:21:45	2018-09-30 01:21

Figure 5-6. *The comments table has been populated with fake data*

You can use the database laravelmodelrelations in your terminal, and by issuing the SELECT * FROM COMMENTS command, you can view how your tables have been populated with the fake data.

```
//code 5.28
mysql> USE laravelmodelrelations
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> SHOW TABLES;
+-----------------------------------+
| Tables_in_laravelmodelrelations |
+-----------------------------------+
| article_tag                     |
| articles                        |
| comments                        |
| countries                       |
| migrations                  |
| password_resets             |
| profiles                    |
| role_user                   |
| roles                       |
| tags                        |
| users                       |
+-----------------------------+
11 rows in set (0.00 sec)

mysql> SELECT * FROM COMMENTS;
```

By issuing such commands, you can also view the corresponding database tables in the terminal. So now you have successfully built all your database tables, and you can concentrate on building your application through controller and views.

Summarizing All Relations

Let's check all the relations one after another. The `Article` controller should be the first candidate, where you will use the `show()` method to display a certain article that has many functionalities attached to it.

You will find out who is the writer of the article and what country the article belongs to. At the same time, you will grab all the articles representing that country. Another important part is the comments section, where you will show all the comments associated with a particular article. Who has written that comment? The username and a link to the user's profile page will also be given.

So, with this article page, you are going to use many features of model relations. One-to-one, one-to-many, many-to-many, has-many-through, and finally polymorphic relationships have been used in this page in one way or other.

You are going to view the full code snippets of `ArticleController.php` (although we have avoided the insert and update parts so far).

```
//code 5.29
//app/HTTP/Controllers/ArticleController.php
<?php
```

```
namespace App\Http\Controllers;

use App\Article;
use App\Country;
use App\User;
use App\Tag;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;

class ArticleController extends Controller
{
    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        $articles = Article::all();
        //$articles = Article::where('active', 1)->orderBy('title',
        'desc')->take(10)->get();
        $users = User::all();
        $tags = Tag::all();
        return view('articles.index', compact('articles', 'users', 'tags'));
    }

    public function main()
    {
        $articles = Article::where('user_id', 1)->orderBy('title', 'desc')-
        >take(4)->get();
        $tags = Tag::all();
        return view('welcome', ['articles' => $articles, 'tags' => $tags]);
    }
    /**
     * Display the specified resource.
     *
     * @param  \App\Article  $article
     * @return \Illuminate\Http\Response
     */
    public function show(Article $article)
    {
        $tags = Article::find($article->id)->tags;
        $article = Article::find($article->id);
        $comments = $article->comments;
        $user = User::find($article->user_id);
        $country = Country::where('id', $user->country_id)->get()->first();

        return view('articles.show', compact('tags','article',
        'country', 'comments', 'user'));
    }
     /**
     * Display the specified resource.
     *
     * @param  \App\Article  $article
     * @return \Illuminate\Http\Response
     */
    public function articles($id)
    {
        $user = User::find($id);

        return view('articles.articles', compact('user'));
    }
}
```

You are interested in the `show()` method currently, so let's view that section of code before you proceed any further.

```
public function show(Article $article)
    {
        $tags = Article::find($article->id)->tags;
        $article = Article::find($article->id);
        $comments = $article->comments;
        $user = User::find($article->user_id);
        $country = Country::where('id', $user->country_id)->get()->first();

        return view('articles.show', compact('tags','article',
        'country', 'comments', 'user'));
    }
```

You should pass every object separately to the `resources/views/articles/show.blade.php` page so that you can use model relations as you want.

The next code snippet is for the `show.blade.php` page. I am going to give all the code, and afterward you will see how it looks in a browser.

```
// code 5.30
// resources/views/articles/show.blade.php
@extends('layouts.app')

@section('content')
<div class="container">
    <div class="row">
        <div class="col-md-8 col-md-offset-2">
            <div class="panel panel-default">
                <div class="panel-heading">
                    <h3 class="pb-3 mb-4 font-italic border-bottom">{{
                    $article->title }}</h3>  by
                    <p>{{ $article->user->name }}</p>
                </div>

                <div class="panel-body">
                    <li class="list-group-item">{{ $article->body }}</li>
                    <strong>Tags:</strong>
                    @foreach($tags as $tag)
                      {{ $tag->tag }}...
                    @endforeach
                </div>
                <p></p>
                <div class="panel panel-default">
                  <h3 class="pb-3 mb-4 font-italic border-bottom">
                All Comments for this Article
              </h3>
              @foreach($user->profile->comments as $comment)
              <li class="list-group-item">{{ $comment->body }}</li>
              <li class="list-group-item">by <strong>
                <a href="/users/{{ $comment->user_id }}">{{ $comment-
                >user->name }}</a>
              </strong></li>
              @endforeach
            </div>
        </div>
    </div>
    <aside class="col-md-4 blog-sidebar">
      <div class="p-3">
          <h3 class="blog-post-title">
            This user belongs to {{ $country->name }}
          </h3>
          <li class="list-group-item-info">Other Articles by
              <p>
                  <a href="/users/{{ $article->user_id }}/articles">{{
                  $article->user->name }}</a>
              </p>
          </li>
          <h3 class="blog-post">
            All articles from {{ $country->name }}
          </h3>
          <hr class="linenums" color="red">
          <div class="panel panel-default">
            <div class="panel-heading">
              @foreach($country->articles as $article)
              <li class="list-group-item">
                <a href="/articles/{{ $article->id }}">
                  {{ $article->title }}
                </a>
              </li>
```

```
                  @endforeach
                </div>
                <hr class="linenums" color="red">
            </div>
          </div>
        </aside>
    </div>
</div>
@endsection
```

Now you can view the main `articles` page, which shows you every feature you have incorporated in your application successfully. From the main `articles` page, you can go to any particular article page and see how it is chained to the `comments`, `users`, and `countries`.

You get everything in this page. The article's title shows up on the top of the page. Next comes the writer's name and the body of the article. After that you have a link that can take you to the other articles written by that writer. After that you have comments. On the right side, you have the writer's name, and you can see to which country the writer belongs. You also have all other articles from that country (Figure 5-5).

You can emulate almost the same techniques to get one user's profile page and show the comments that have been written over time.

```
// code 5.31
// app/HTTP/Controllers/UserController.php
<?php

namespace App\Http\Controllers;


use App\User;
use App\Profile;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;

use RepositoryDB\DBUserRepository as DBUserRepository;
```

```
class UserController extends Controller
{
    public $users;

    public function __construct(DBUserRepository $users) {

        $this->users = $users;

    }
    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        $users = $this->users->all();
        return view('users.index', compact('users'));
    }

    /**
     * Show the form for creating a new resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function create()
    {
        //
    }

    /**
     * Store a newly created resource in storage.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
```

```
public function store(Request $request)
{
    //
}

/**
 * Display the specified resource.
 *
 * @param  \App\User  $user
 * @return \Illuminate\Http\Response
 */
public function show(User $user)
{
  //$roles = User::find($user->id)->roles;
    $user = User::find($user->id);
    return view('users.show', compact('user'));
}

/**
 * Show the form for editing the specified resource.
 *
 * @param  \App\User  $user
 * @return \Illuminate\Http\Response
 */
public function edit(User $user)
{
    //
}

/**
 * Update the specified resource in storage.
 *
 * @param  \Illuminate\Http\Request  $request
 * @param  \App\User  $user
 * @return \Illuminate\Http\Response
 */
```

```
    public function update(Request $request, User $user)
    {
        //
    }

    /**
     * Remove the specified resource from storage.
     *
     * @param  \App\User  $user
     * @return \Illuminate\Http\Response
     */
    public function destroy(User $user)
    {
        //
    }
}
```

The code snippets of `UserController.php` are simple enough, although you have passed only one user object to `show.blade.php`, through the `show()` method. This handles many complex queries like getting all the comments that have been posted on that particular page.

```
// code 5.32
// resources/views/users/show.blade.php
@extends('layouts.app')

@section('content')
<div class="container">
    <div class="row">
        <div class="col-md-10 col-md-offset-2">
            <div class="panel panel-default">
                <div class="panel-heading">
                  <h3 class="pb-3 mb-4 font-italic border-bottom">
                    Profile of {{ $user->name }}
                  </h3>
                </div>
                <div class="panel-body">
                  <strong>Name : </strong>
```

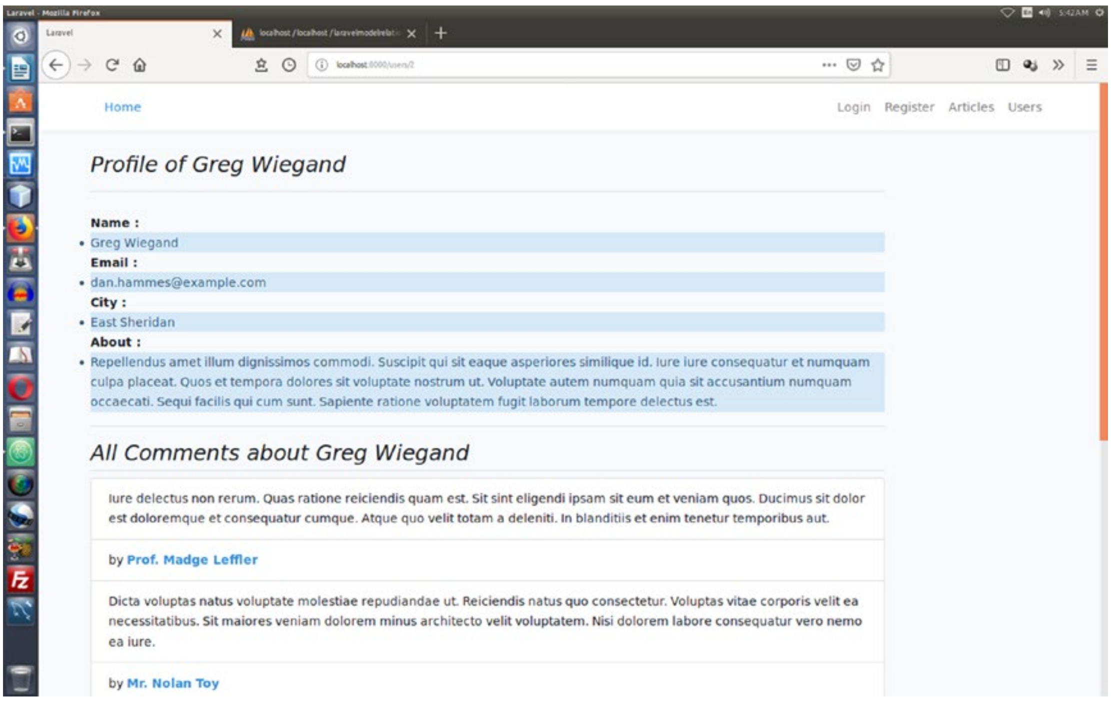
```
                    <li class="list-group-item-info">{{ $user->name }}</li>
                    <strong>Email : </strong>
                    <li class="list-group-item-info">{{ $user->email }}</li>
                    <strong>City : </strong>
                    <li class="list-group-item-info">{{ $user->profile->
                    city }}</li>
                    <strong>About : </strong>
                    <li class="list-group-item-info">{{ $user->profile->
                    about }}</li>
                </div>
                <div class="panel panel-default">
                    <hr>
                </div>
                <h3 class="pb-1 mb-2 font-italic border-bottom">
                  All Comments about {{$user->name}}
                </h3>
                <div class="panel panel-default">
                  @foreach($user->profile->comments as $comment)
                  <li class="list-group-item">{{ $comment->body }}</li>
                  <li class="list-group-item">by <strong>
                    <a href="/users/{{ $comment->user_id }}">{{ $comment-
                    >user->name }}</a>
                  </strong></li>
                  @endforeach
                </div>
                <div class="panel panel-default">
                    <hr>
                </div>
            </div>
        </div>
    </div>
</div>
@endsection
```

In the previous code snippet (code 5.61), the most interesting part is this:

```
@foreach($user->profile->comments as $comment)
                  <li class="list-group-item">{{ $comment->body }}</li>
                  <li class="list-group-item">by <strong>
                    <a href="/users/{{ $comment->user_id }}">{{ $comment-
                     >user->name }}</a>
                  </strong></li>
                  @endforeach
```

You set up a user profile by using a one-to-one relationship, and afterward you connect that profile object to all the comments that have been posted to that page. Using the `comment` object, you again go back to fetch the username associated with it.

Now you can display any user's profile page in a unique way (Figure 5-7).


Laravel - Mozilla Firefox
Laravel
localhost / localhost / laravelmodelrelati
localhost:8000/users/2
5:42AM
Home
Login
Register
Articles
Users

Profile of Greg Wiegand

Name :
- Greg Wiegand

Email :
- dan.hammes@example.com

City :
- East Sheridan

About :
- Repellendus amet illum dignissimos commodi. Suscipit qui sit eaque asperiores similique id. Iure iure consequatur et numquam culpa placeat. Quos et tempora dolores sit voluptate nostrum ut. Voluptate autem numquam quia sit accusantium numquam occaecati. Sequi facilis qui cum sunt. Sapiente ratione voluptatem fugit laborum tempore delectus est.

All Comments about Greg Wiegand

Iure delectus non rerum. Quas ratione reiciendis quam est. Sit sint eligendi ipsam sit eum et veniam quos. Ducimus sit dolor est doloremque et consequatur cumque. Atque quo velit totam a deleniti. In blanditiis et enim tenetur temporibus aut.

by **Prof. Madge Leffler**

Dicta voluptas natus voluptate molestiae repudiandae ut. Reiciendis natus quo consectetur. Voluptas vitae corporis velit ea necessitatibus. Sit maiores veniam dolorem minus architecto velit voluptatem. Nisi dolorem labore consequatur vero nemo ea iure.

by **Mr. Nolan Toy**

***Figure 5-7.** User's profile page*

CHAPTER 6

Handling User Data and Redirects

A *redirect* is a kind of response; it's part of a request-response cycle in Laravel. Therefore, Laravel has designed it as a `RedirectResponse` instance or object that you can use through the global helper function `redirect()`. It comes from `Illuminate\Http\RedirectResponse`. You need it for many reasons. One of them is to validate a user's data or input. If the user input is invalid, it is a responsibility of a well-designed application to send the user a message and return to the original state.

If you want to redirect to the previous location, another good method is the `back()` global helper. You will learn about both in this chapter. I will first cover how the redirect methods work in Laravel and how you can handle user data efficiently.

After that, I will show how to build an administrative dashboard in the news article application that you've been working on.

So far, you have seen how to build a dynamic database-driven CRUD application using a single administrative power. Later in this chapter, you will learn more about web forms, validation, and finally how to build an administrative dashboard. The full application code is available with the download for the book.

In Chapter 8, you will switch to another application, a company/project/task management application, and learn more about role-based authentication, middleware, and authorization. You'll make that application role-based. In the news article application in this chapter, though, you will deal with one administrator role. In the next application, you will have project managers and general members along with the administrator role, and they will have their own pages based on their roles. The general members will not have administrative powers like the other two.

In Laravel there are several methods to implement authentication; in the news application, you saw one method already. In Chapter 8, you will see the others.

© Sanjib Sinha 2019
S. Sinha, *Beginning Laravel*, https://doi.org/10.1007/978-1-4842-4991-8_6

How Redirect Methods Work

There are some helper functions you need when you want to redirect users to a certain location. Suppose there is a comment form where a user does not fill in some required fields. In such cases, the submitted form is invalid, and the user must be redirected to the previous location.

```
//code 6.1
    Route::post('user/profile', function () {
        // Validate the request...
        return back()->withInput();
    });
```

Besides the `back()` helper method, you can use the global `redirect()` method, and you can use it anywhere in your application.

When you use Laravel form request validation, Laravel will redirect you with errors. The given inputs are kept within the chained `back()->withInput()` method. You can also write the previous code (6.1) like this:

```
Route::post('user/profile', function () {
        // Validate the request...
        return Redirect::back()->withInput(Input::all());
    });
```

Instead of global helper methods, you are using the `Redirect` and `Input` class variables directly. In such cases, Request comes from `Illuminate\Foundation\Validation\ValidatesRequest`.

Consider your `ArticleController`. In the `index` method, you are using the `redirect()` helper to make it sure that the user is an administrator, as shown here:

```
//code 6.2
//app/Http//Controller/ArticleController.php
    public function index()
    {
        //
        if(Auth::user()->is_admin == 1){
        $articles = Article::orderBy('published_at', 'asc')->get();
        return view('articles.index', compact('articles'));
      }
      else {
        return redirect('home');
      }
    }
```

From the Illuminate\Http\RedirectResponse class, you get the instances of redirect responses. The class should contain the proper headers to redirect users to another URL. There are several ways to generate a RedirectResponse instance. The simplest one is of course to use the global redirect helper.

On the Redirector instance, you can call any method like route() to send the user to another URL. Redirector is another class that comes from Illuminate\Routing, and it generates many useful global helper methods. When your home page is secured, you can safely redirect the user to home as it will by default choose the login page.

```
//code 6.3
return redirect()->route('login');
```

Let's assume that your route has parameters; in such cases, you may pass them as the second argument to the route method, as shown here:

```
//code 6.4
// For a route with the following URI: article/{id}
return redirect()->route('article', ['id' => 1]);
```

Redirecting to a controller action is one of the most common cases. You can pass the controller and action name to the action method.

```
//code 6.5
return redirect()->action('ArticleController@index');
```

If your controller needs parameters, you can pass them as the second argument to the action method.

```
//code 6.6
return redirect()->action(
        'ArticleController@show', ['id' => 1]
    );
```

You mostly use the `redirect` global helper for flashing session data.

Note You use the session to store data because HTTP-driven applications are stateless. When you flash session data, the data is saved for only the subsequent HTTP request. Laravel provides this support automatically.

When you edit an article and redirect to a new URL, you may want to flash data to a session. This is usually done at the same time because Laravel validates the data and makes it ready for either saving in the database or returning as a status message, and you want that data at the new URL. The following example explains it. You flash the success message after performing the required actions.

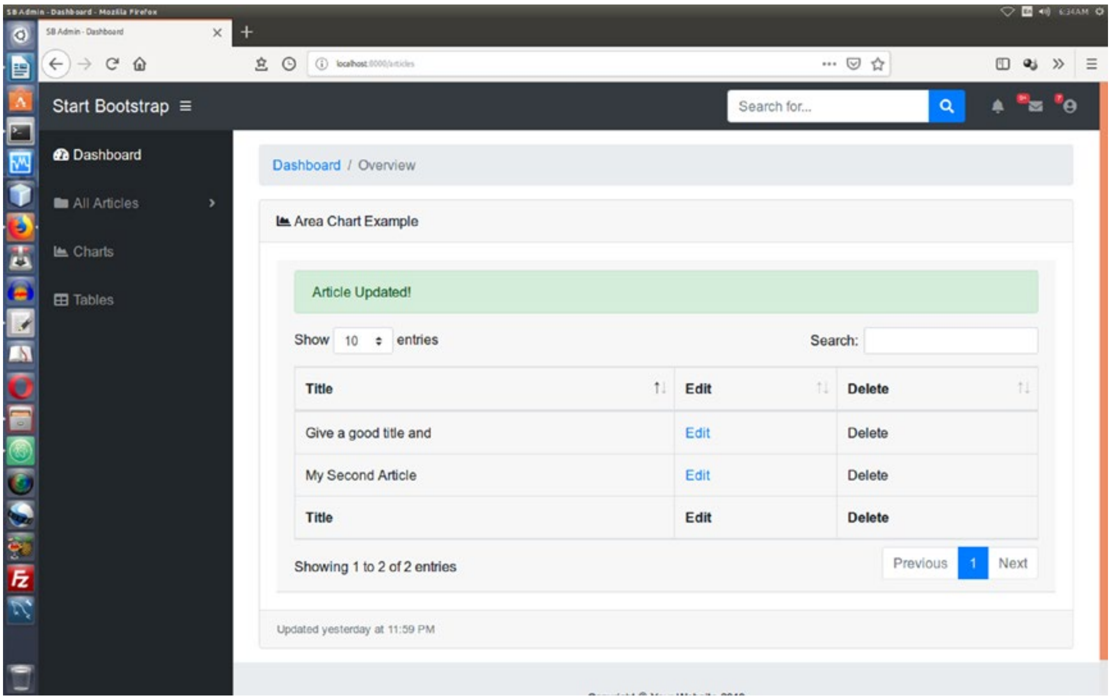
Consider the `ArticleController` `update()` method shown here:

```
//code 6.7
    public function update(Request $request, $id)
    {
        //
        if(Auth::user()->is_admin == 1){

          if($file = $request->file('image')){

              $name = $file->getClientOriginalName();

              $post = Article::findOrFail($id);
              $post->title = $request->input('title');
              $post->body = $request->input('body');
              $post->published_at = $request->input('published_at');
              $post->image = $name;
              $post->save();

              $file->move('images/upload', $name);
          }
        else {
            // code...
            $post = Article::findOrFail($id);
            $post->title = $request->input('title');
            $post->body = $request->input('body');
            $post->published_at = $request->input('published_at');
            $post->save();

        }

        if($post){
            return redirect('articles')->with('status', 'Article
            Updated!');
        }
      }
  }
```

It ends with the flashed message from the session. Now, in the articles `index.blade.php` page, you can flash that message like this:

```
//code 6.8
//resources/views/articles/index.blade.php
    @if (session('status'))
        <div class="alert alert-success">
            {{ session('status') }}
        </div>
    @endif
```

When you use any article, it displays the flash-data, as shown in Figure 6-1.



Article Updated!

Show 10 entries Search:

Title	Edit	Delete
Give a good title and	Edit	Delete
My Second Article	Edit	Delete
Title	Edit	Delete

Showing 1 to 2 of 2 entries Previous 1 Next

Updated yesterday at 11:59 PM

Figure 6-1. *Article updated with flash-data*

What Is a Request Object?

On any controller method, you can type-hint the `Illuminate\Http\Request` class. The incoming request instance will automatically be injected by the service container.

Consider the `ArticleController` `store()` method shown here:

```
//code 6.9
    public function store(Request $request)
    {
        if($file = $request->file('image')){
         $name = $file->getClientOriginalName();
         $post = new Article;
         $post->title = $request->input('title');
         $post->body = $request->input('body');
         $post->published_at = $request->input('date');
         $post->image = $name;
        $post->save();

  $file->move('images/upload', $name);

   }

        if($post){
            return redirect('articles')->with('status', 'Article Created!');
        }
    }
```

You create new articles by using the incoming requests from the form inputs, like this:

```
$post = new Article;
$post->title = $request->input('title');
//code incomplete
```

In the same way, in the `edit()` method, you can use the dependency injection and route parameters to edit any existing article.

In such cases, your controller method is also expecting input from a route parameter. You can still type-hint `Illuminate\Http\Request` and access the route parameter ID by defining the controller `update()` method like this:

```
//code 6.10
  /**
     * Update the specified resource in storage.
     *
     * @param  \Illuminate\Http\Request $request
     * @param  int $id
     * @return \Illuminate\Http\Response
     */
    public function update(Request $request, $id)
    {
        //
        if(Auth::user()->is_admin == 1){
          if($file = $request->file('image')){

              $name = $file->getClientOriginalName();

              $post = Article::findOrFail($id);
              $post->title = $request->input('title');
              $post->body = $request->input('body');
              $post->published_at = $request->input('published_at');
              $post->image = $name;
              $post->save();

              $file->move('images/upload', $name);
   }
   else {
     // code...
     $post = Article::findOrFail($id);
     $post->title = $request->input('title');
     $post->body = $request->input('body');
     $post->published_at = $request->input('published_at');
     $post->save();

  }
      if($post){
             return redirect('articles')->with('status', 'Article Updated!');
         }
    }
    }
```

There are two parameters; one is `\Illuminate\Http\Request $request`, and another is `int $id`.

How Requests and Responses Work

The full `\Illuminate\Http\Response` instances can contain anything. I'll often return a full view. By default Laravel 5.8 allows you to generate redirects to controller actions; this means through controller actions you can return a view page. You don't even have to use the full namespace. Laravel's `RouteServiceProvider` automatically resolves it.

Whenever the user sends a request, the route or controller should return a response to be returned to the user's browser.

In that sense, I have already covered the main part of `response` instance in the previous `redirect` section. Now Laravel provides several different ways to return responses.

The most basic one is a string from a route or a controller; the framework automatically converts a string to an HTTP response. You can also return an array that Laravel converts into a JSON response. This is good for creating APIs. At the same time, based on this mechanism, you can handle complex database records.

```
//code 6.11
Route::get('/', function () {
    return [1, 2, 3];
});
```

The same way you pass Eloquent collections from your route or controller, they will also be automatically converted to JSON.

Thankfully, Laravel lets you return not only simple strings or arrays from the route or controller; the `Illuminate\Http\Response` instance or object is extremely powerful and can return a full view blade seamlessly. You will see this in the next sections.

Introducing Validation

There are several different approaches to validate an application's incoming data. By default Laravel's base controller class uses a `ValidatesRequests` trait that provides a convenient method to validate incoming HTTP requests with a variety of powerful validation rules.

Let's see the different ways of validating requests one after another.

You may assume your `ArticleController` is resourceful. Or, you can use routes like this:

```
//code 6.12
Route::get('article/create', 'ArticleController@create');
Route::post('article', 'ArticleController@store');
```

As usual, the GET route will display the form to create articles, and the POST route will store the contents in your database.

In such cases, ArticleController uses the default validation rules in the store method like this:

```
//code 6.13
 /**
     * Store a new article content.
     *
     * @param  Request  $request
     * @return Response
     */
    public function store(Request $request)
    {
        $validatedData = $request->validate([
            'title' => 'required|unique:articles|max:255',
            'body' => 'required',
        ]);
        // The content is valid...
    }
```

The logic is simple for the body of the content, and it is required. For the title you use some more flags, such as the title should not exceed 255 characters, the unique rule must be maintained, and so on. If the validation rules pass, the controller will continue functioning and execute in a normal way. If the validation fails, it will give you a proper response, and the user can view it in the respective view template.

In the previous code, you can add more functionality such as validation attributes so that if the first attribute fails, then the next one will not work anymore.

```
//code 6.14
 <?php

    namespace App\Http\Controllers;

    use Illuminate\Http\Request;
    use App\Http\Controllers\Controller;

    class ArticleController extends Controller
    {
        /**
         * Show the form to create a new blog post.
         *
         * @return Response
         */
        public function create()
        {
            return view('article.create');
        }

        /**
         * Store a new blog post.
         *
         * @param  Request  $request
         * @return Response
         */
        public function store(Request $request)
        {
            $validatedData = $request->validate([
            'title' => 'bail|required|unique:articles|max:255',
            'body' => 'required',
        ]);
        // The content is valid...
        }
    }
```

In the previous code, this line is the important one:

```
'title' => 'bail|required|unique:articles|max:255',
```

Here you assign the `bail` rule to your attribute, and if the `unique` rule on the `title` attribute fails, the `max` rule will not be checked. If you want to make your application slightly faster, you may use this feature because it doesn't check all the rules.

Now, the question is, how will these errors be flashed? Well, Laravel has taken care of this functionality automatically. All you need to do is add these lines of code in your `create.blade.php` file:

```
//code 6.15
<!-- /resources/views/article/create.blade.php -->

    <h1>Create Article</h1>

    @if ($errors->any())
        <div class="alert alert-danger">
            <ul>
                @foreach ($errors->all() as $error)
                    <li>{{ $error }}</li>
                @endforeach
            </ul>
        </div>
    @endif
```

Here it is extremely important to note that the `$errors` variable is bound to the view by the `Illuminate\View\Middleware\ShareErrorsFromSession` middleware, which is provided by the `web` middleware group. So, this `$errors` variable is always present in your view template. You don't have to define it explicitly. The `$errors` variable is always defined and can be safely used.

Now, what happens when the request fields are `nullable`? You don't want to consider the `null` values as invalid. Laravel has thought about this already and includes the `TrimStrings` and `ConvertEmptyStringsToNull` middleware in your application's global middleware stack. These middlewares are listed in the stack by the `App\Http\Kernel` class. This gives you more choices; for that reason, you will often need to mark your `optional` request fields as `nullable`. In such cases, you do not want the validator to consider `null` values as invalid.

Consider this code:

```
//code 6.16
$request->validate([
        'title' => 'bail|required|unique:articles|max:255',
        'body' => 'required',
        'publish_at' => 'nullable|date',
    ]);
```

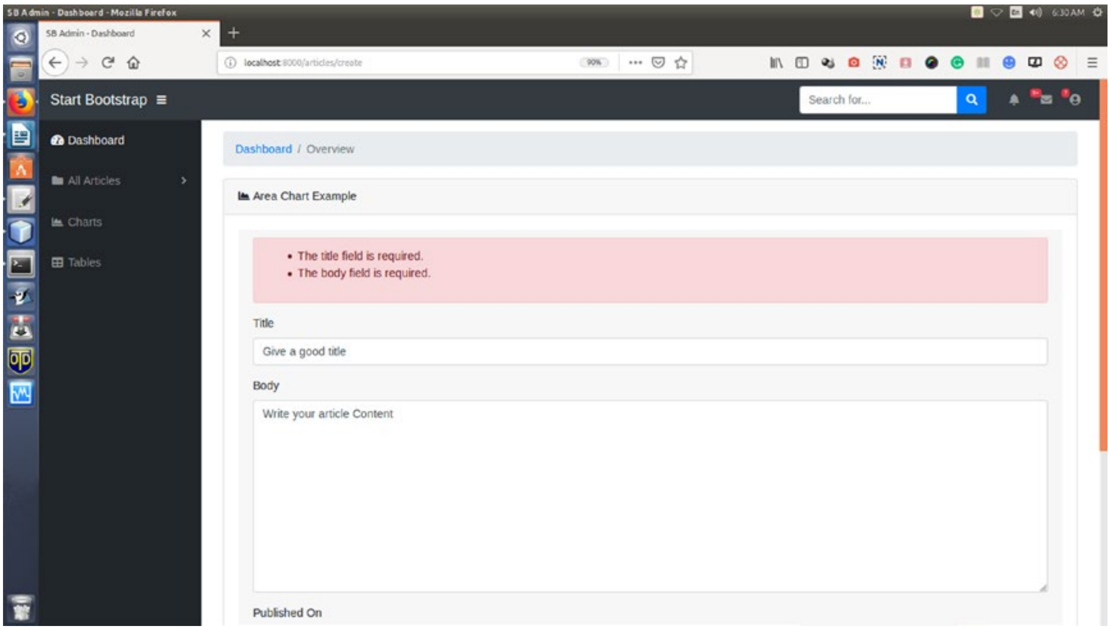
In this code, you are mentioning categorically that the `publish_at` field may be either `null` or a valid date representation. If you do not add the `nullable` modifier to the rule definition, the validator will have considered `null` an invalid date.

Consider the `ArticleController` logic you have used so far. In addition, take a look at the following `create.blade.php` file as an example to see how the validation works:

```
//code 6.17
//app/HTTP/Controllers/ArticleController.php
    public function store(Request $request)
    {
        //
        $validatedData = $request->validate([
            'title' => 'required|unique:articles|max:255',
            'body' => 'required',
        ]);
        if($file = $request->file('image')){

        $name = $file->getClientOriginalName();

        $post = new Article;
        $post->title = $request->input('title');
        $post->body = $request->input('body');
        $post->published_at = $request->input('date');
        $post->image = $name;

        $post->save();

        $file->move('images/upload', $name);

   }
```

```
        if($post){
            return redirect('articles');
        }
    }
```

If you try to submit the form without the title and body fields, you get the response shown in Figure 6-2.



- The title field is required.
- The body field is required.

Title

Give a good title

Body

Write your article Content

Published On

Figure 6-2. *Displaying the error messages on the create.blade.php view template*

You may want to create your own validate method on the request. In that case, you may use the Validator facade, and the make method on the facade generates a new validator instance.

Let's see how it works in your ArticleController:

```
//code 6.18
//app/HTTP/Controllers/ArticleController.php
class ArticleController extends Controller
    {
        public function store(Request $request)
        {
            $validator = Validator::make($request->all(), [
                'title' => 'required|unique:articles|max:255',
                'body' => 'required',
            ]);

            if ($validator->fails()) {
                return redirect('article/create')
                            ->withErrors($validator)
                            ->withInput();
            }
            //code...
        }
    }
```

This will give you the same effect you saw in Figure 6-2. It entirely depends on the developer's strategy which logic will be used, but there is no hard-and-fast rule that one is more advantageous than the other.

Web Form Fundamentals

Laravel provides an easy method to use Forms and HTML in your application, and it, at the same time, helps you protect your site from cross-site request forgeries. I will talk about this in a minute, but, before that, I will introduce two separate methods of using forms.

There are two ways you can use forms in Laravel. First is the traditional method of using form tags like you use generally, as shown here:

```
//code 6.19
<form method="POST" action="{{ route('login') }}">
@csrf
```

The second way is to use the `laravelcollective/html` package, as shown here:

```
//code
{!! Form::open(['url' => 'foo/bar']) !!}
    //
{!! Form::close() !!}
```

If you want to use the traditional form input methods, it is perfectly okay. If you want to use the second method using the Laravel `html` package, then you need to add a few lines of code as new `provider` and `class` aliases.

Let's first see how you can handle Laravel `html` and `form` packages. After that, you will get a sneak preview of the traditional form input methods whose syntaxes are completely different. However, both work seamlessly with Laravel, so what you will use entirely depends on you.

Using the Laravel HTML and Form Packages

First, you need to install the packages. Open your terminal and issue this command:

```
//code 6.20
composer require "laravelcollective/html":"^5.4.0"
```

Next, you need to add your new provider to the `providers` array of `config/app.php`, as shown here:

```
//code 6.21
  'providers' => [
    // ...
    Collective\Html\HtmlServiceProvider::class,
    // ...
  ],
```

Finally, you need to add two class aliases to the `aliases` array of `config/app.php`:

```
//code 6.22
  'aliases' => [
    // ...
      'Form' => Collective\Html\FormFacade::class,
      'Html' => Collective\Html\HtmlFacade::class,
    // ...
  ],
```

Suppose you need to create an article resource. The creation code in `resources/views/articles/create.blade.php` will look like this:

```
//code 6.23
//resources/views/articles/create.blade.php
      <div class="card-header">

{!! Form::open(['url' => 'articles', 'files' => true]) !!}
<div class="form-group">
{!! Form::label('title', 'Title', ['class' => 'awesome']) !!}
{!! Form::text('title', 'Give a good title', ['class' => 'form-control']) !!}
</div>
<div class="form-group">

{!! Form::label('body', 'Body', ['class' => 'awesome']) !!}
{!! Form::textarea('body', 'Write your article Content', ['class' => 'form-
control']) !!}
</div>
<div class="form-group">
{!! Form::label('published_at', 'Published On', ['class' => 'awesome']) !!}
{!! Form::input('date', 'published_at', null, ['class' => 'form-control']) !!}
</div>
<div class="form-group">
{!! Form::file('image') !!}
</div>
<div class="form-group">
{!! Form::submit('Add Article', ['class' => 'btn btn-primary form-control']) !!}
</div>
{!! Form::close() !!}
</div>
```

Here, the first line is important to understand how the Laravel `html` and `form` packages work. Consider this line:

```
{!! Form::open(['url' => 'articles', 'files' => true]) !!}
```

Here it is assumed that the method is POST; however, you could have done it like this:

```
{!! Form::open(array('url' => 'foo/bar', 'method' => 'put', 'files' => true)) !!}
```

This says you are planning to upload an image for which you have used another key/value pair of file options like this: `files` => `true`. This means your forms will accept the file upload.

You can also open forms that point to the named routes or controller actions like these instances:

```
//code 6.24
{!! Form::open(array('route' => 'route.name')) !!}
{!! Form::open(array('action' => 'Controller@method')) !!}
```

The form fields look like Figure 6-3 in a browser.

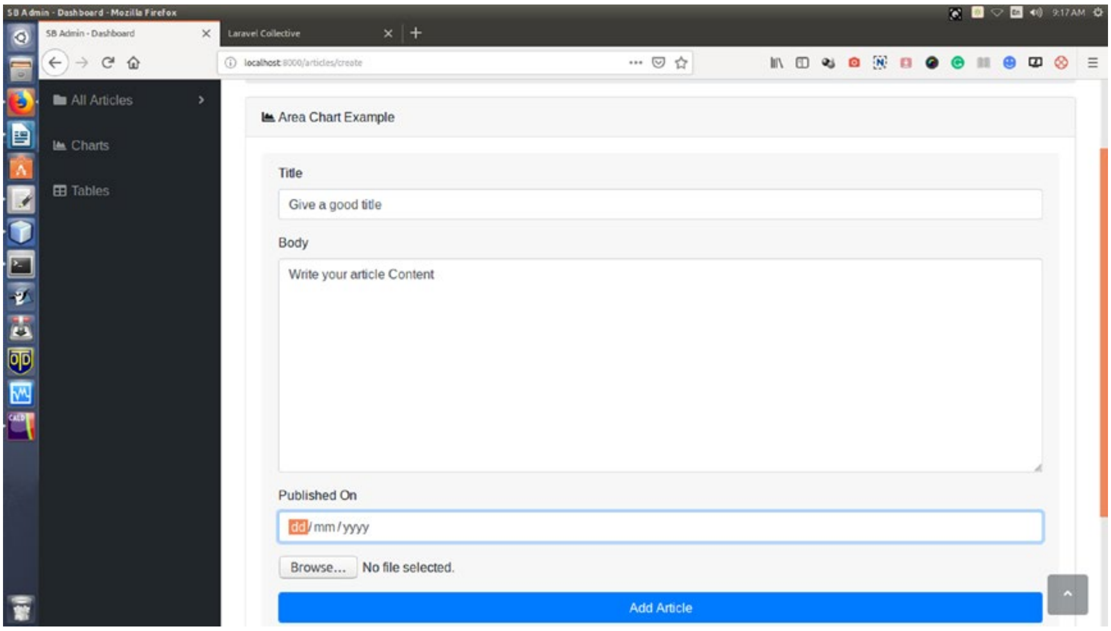


***Figure 6-3.** Showing the form for creating the articles resource, with the Laravel html/form packages working in the background*

In your file `ArticleController.php`, you have two methods associated with this form, as shown here:

```
//code 6.25
//app/HTTP/Controllers/ArticleController.php
    public function create()
    {
        if(Auth::user()->is_admin == 1){
            return view('articles.create');
        }
        else {
          return redirect('home');
        }
    }
    public function store(Request $request)
    {
        if($file = $request->file('image')){
        $name = $file->getClientOriginalName();
  $post = new Article;
  $post->title = $request->input('title');
  $post->body = $request->input('body');
  $post->published_at = $request->input('date');
  $post->image = $name;
  $post->save();
  $file->move('images/upload', $name);
   }
```

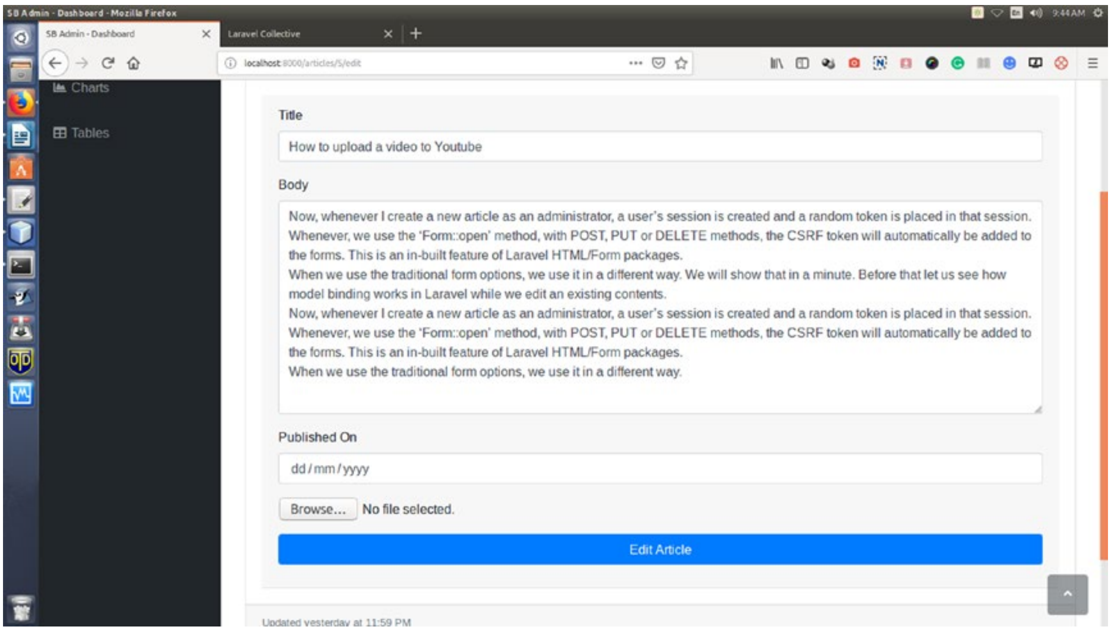
You have already seen how `Request` objects work in Laravel. At the same time, you can upload the image to the `images/upload` folder.

Now, whenever you create a new article as an administrator, a user session is created, and a random token is placed in that session. Whenever you use the `Form::open` method, with POST, PUT, or DELETE methods, the CSRF token will automatically be added to the forms. This is built-in feature of the Laravel `html` and `form` packages.

When you use the traditional form options, you use them in a different way. I will show that in a minute. Before that, let's see how model binding works in Laravel while you edit existing contents.

Model Binding

Suppose you want to edit an existing article that has an ID of 5. In a browser, it looks like Figure 6-4.


SB Admin - Dashboard - Mozilla Firefox
SB Admin - Dashboard
Laravel Collective
localhost:8000/articles/5/edit
Charts
Tables
Edit Article

Title

How to upload a video to Youtube

Body

Now, whenever I create a new article as an administrator, a user's session is created and a random token is placed in that session. Whenever, we use the 'Form::open' method, with POST, PUT or DELETE methods, the CSRF token will automatically be added to the forms. This is an in-built feature of Laravel HTML/Form packages.
When we use the traditional form options, we use it in a different way. We will show that in a minute. Before that let us see how model binding works in Laravel while we edit an existing contents.
Now, whenever I create a new article as an administrator, a user's session is created and a random token is placed in that session. Whenever, we use the 'Form::open' method, with POST, PUT or DELETE methods, the CSRF token will automatically be added to the forms. This is an in-built feature of Laravel HTML/Form packages.
When we use the traditional form options, we use it in a different way.

Published On

dd / mm / yyyy

Browse... No file selected.

Updated yesterday at 11:59 PM

Figure 6-4. *Showing the form for editing the existing articles resource*

You can update this article with new materials, updating everything including the image and the publishing dates.

Let's see how model binding works; here is the code for the `resources/views/articles/edit.blade.php` page:

```
//code 6.26
//resources/views/articles/edit.blade.php
<div class="card-header">
{!! Form::model($article, ['route' => ['articles.update', $article->id],
'method' => 'PUT', 'files' => true]) !!}

<div class="form-group">
{!! Form::label('title', 'Title', ['class' => 'awesome']) !!}
{!! Form::text('title', "$article->title", ['class' => 'form-control']) !!}
</div>
```

```
<div class="form-group">
{!! Form::label('body', 'Body', ['class' => 'awesome']) !!}
{!! Form::textarea('body', "$article->body", ['class' => 'form-control']) !!}
</div>
<div class="form-group">
{!! Form::label('published_at', 'Published On', ['class' => 'awesome']) !!}
{!! Form::input('date', 'published_at', null, ['class' => 'form-control']) !!}
</div>
<div class="form-group">
{!! Form::file('image') !!}
</div>
<div class="form-group">
{!! Form::submit('Edit Article', ['class' => 'btn btn-primary form-
control']) !!}
</div>
{!! Form::close() !!}
</div>
```

You actually want to populate the form based on the contents of the `Article` model. For that reason, you use the `Form::model` method, which in turn populates the input fields with the existing data based on the resource ID.

Whenever you generate a form element like a text input field, the model's value matching the field's name is automatically set as the field value. At the same time, let's see how the `ArticleController` methods work in this scenario.

```
//code 6.27
//app/HTTP/Controllers/ArticleController.php
    public function edit($id)
    {
        if(Auth::user()->is_admin == 1){
        $article = Article::findOrFail($id);
        return view('articles.edit', compact('article'));
      }
      else {
        return redirect('home');
      }
    }
    public function update(Request $request, $id)
    {
        if(Auth::user()->is_admin == 1){

          if($file = $request->file('image')){

              $name = $file->getClientOriginalName();

              $post = Article::findOrFail($id);
  $post->title = $request->input('title');
  $post->body = $request->input('body');
  $post->published_at = $request->input('published_at');
  $post->image = $name;
  $post->save();
                  $file->move('images/upload', $name);
   }
   else {
     $post = Article::findOrFail($id);
  $post->title = $request->input('title');
  $post->body = $request->input('body');
  $post->published_at = $request->input('published_at');
  $post->save();
  }
      if($post){
             return redirect('articles');
         }
    }
    }
```

You have already seen how text and text area fields work in forms. For a password field, you can use this:

```
//code 6.28
{!! Form::password('password') !!}
```

In the same way, you can generate other inputs in this way:

```
{!! Form::email($name, $value = null, $attributes = array()) !!}
{!! Form::file($name, $attributes = array()) !!}
```

For checkboxes and radio buttons, you can use this:

```
{!! Form::checkbox('name', 'value') !!}
{!! Form::radio('name', 'value') !!}
```

You can generate a checkbox or radio input that is checked as follows:

```
{!! Form::checkbox('name', 'value', true) !!}
{!! Form::radio('name', 'value', true) !!}
```

For drop-down lists, you generally generate them in this way:

```
{!! Form::select('size', array('L' => 'Large', 'S' => 'Small')) !!}
```

You can generate a drop-down list with a selected default as follows:

```
{!! Form::select('size', array('L' => 'Large', 'S' => 'Small'), 'S') !!}
```

For generating a submit button, the process is quite simple, as shown here:

```
echo Form::submit('Click Me!');
```

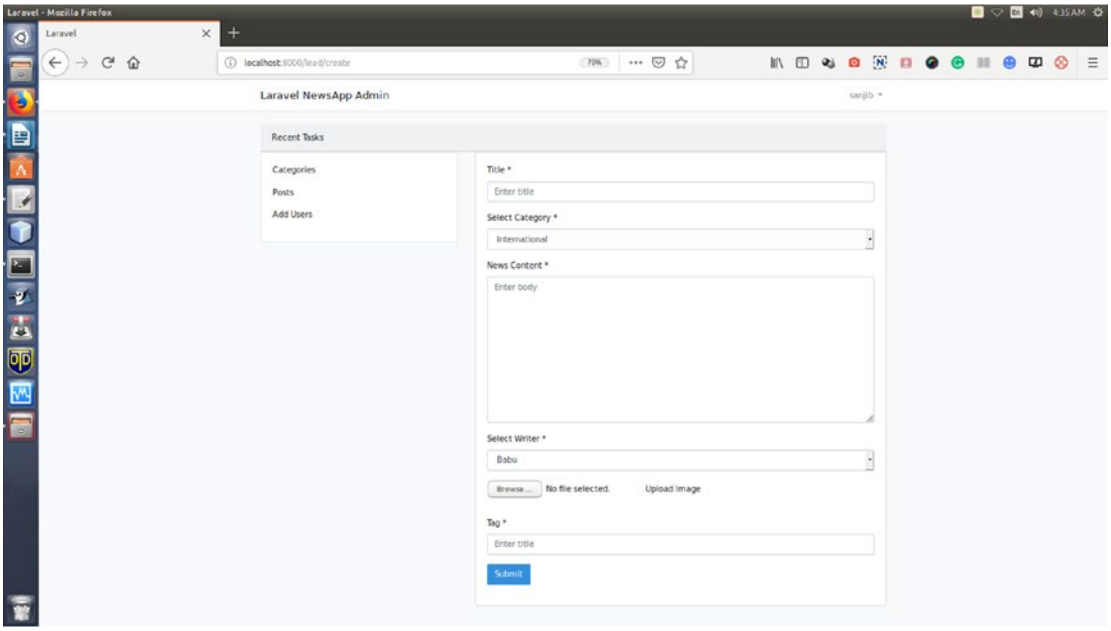
For full lists, please view `https://laravelcollective.com/`. You can get the updated form inputs there.

The Traditional Way of Form Inputs

It is not mandatory that you have to use the Laravel HTML packages. You can follow the traditional form input methodology, and you will get the same result.

In this section, you will see two instances of the traditional approach. In the first one, you will see how you can use the form inputs to create content.

I have used the select options to choose from different categories here. See Figure 6-5.



Title *
Enter title
Select Category *
International
News Content *
Enter body
Select Writer *
Babu
No file selected.
Tag *
Enter title

Figure 6-5. Displaying the form to create contents

Figure 6-5 shows that you will be adding `title`, `body`, and `tag` elements for your `articles`. You will also upload an image and select a category from various categories.

The code looks like this in your new `article.create` Blade page:

```
//code 6.29
@extends('layouts.app')

@section('content')
<div class="container">
  <div class="card-header">Recent Tasks</div>

    <div class="row justify-content-left">
        <div class="col-md-4">
            <div class="card">
              <div class="card-body">
                Categories
                <p></p>
                Articles
                <p></p>
                Add Users
                <p></p>

                </div>
            </div>
        </div>
        <div class="col-md-8">
            <div class="card">
              <div class="card-body">
                <form enctype="multipart/form-data" method="post"
                action="{{ route('article.store') }}">
                          {{ csrf_field() }}
                          <div class="form-group">
                             <label for="post-name">Title
                               <span class="required">*</span>
                             </label>
<input   placeholder="Enter title" id="post-title" required name="title"
spellcheck="false" class="form-control"/>
                              </div>
                                 @if($categories == null)
<input class="form-control" type="hidden" required name="category_id"
value="{{ $category_id }}"/>
                          </div>
                                 @endif
                                      @if($categories != null)
                                      <div class="form-group">
                                          <label for="category-
                                          content">Select Category</label>
                                          <span class="required">*</span>
                                          <select name="category_id"
                                          class="form-control" >
                                          @foreach($categories as
                                          $category)
                                                  <option value=
                                                  "{{$category->id}}">
```

```
                                                          {{$category->name}}
                                                        </option>
                                                      @endforeach
                                               </select>
                                           </div>
                                           @endif
                                  <div class="form-group">
                                                  <label for="project-
                                                  content">News Content</label>
                                                  <span class="required">*</span>
                                                  <textarea placeholder="Enter
                                                  body"
                                                            style="resize:
                                                            vertical"
                                                            id="post-body"
                                                            required
                                                            name="body"
                                                            rows="10"
                                                            spellcheck="false"
                                                            class="form-control
                                                            autosize-target text-
                                                            left">
                                                   </textarea>
                                              </div>
                                              @if($writers == null)
                                              <input class="form-control"
                                              type="hidden" required
                                              name="writer_id"
                                              value="{{ $writer_id }}"/>
                                              </div>
                                              @endif
                                                   @if($writers != null)
                                                   <div class="form-group">
                                                       <label for="category-
                                                       content">Select Writer</
                                                       label>
```

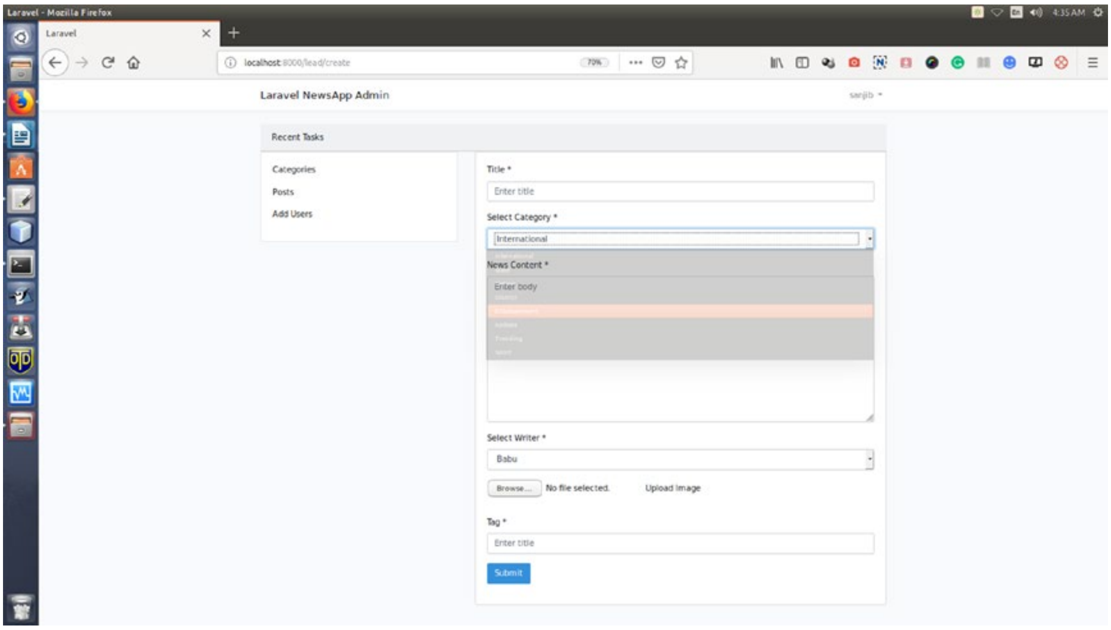
```
                                   <span class="required">*</
                                   span>
                                   <select name="writer_id"
                                   class="form-control" >
                                   @foreach($writers as
                                   $writer)
                                           <option value="{{
                                           $writer->id }}">
                                             {{ $writer->name
                                             }}
                                           </option>
                                         @endforeach
                                   </select>
                               </div>
                               @endif
                          <div class="form-group">
                             <label class="form-group">
                                 <input type="hidden"
                                 name="MAX_FILE_SIZE"
                                 value="3000000" />
                                 <input name="image"
                                 type="file">
                                <span class="custom-file-
                                control">Upload Image</span>
                              </label>
                         </div>
                         <div class="form-group">
                            <label for="post-name">Tag
                              <span class="required">*</span>
                            </label>
<input   placeholder="Enter title" id="post-title" required
name="tag" spellcheck="false" class="form-control"/>
                             </div>
                  <div class="form-group">
```

```
                                                <input type="submit" class="btn btn-primary"
                                                        value="Submit"/>
                                           </div>
            </form>
                     </div>
                </div>
           </div>
      </div>
</div>
@endsection
```

This code is long, but let's concentrate on the first line:

```
<form enctype="multipart/form-data" method="post" action="{{
route('article.store') }}">
                                {{ csrf_field() }}
```

You can select any category, as shown in Figure 6-6.



Select Category * International

Select Writer * Babu

Figure 6-6. *Selecting categories while creating contents*

The action is `article.store`. This means in the `ArticleController.php` file you need to have some extra logic like the following; this will add a `tag` and `category` and allow you to upload an image; the `title` and `body` were there:

```
//code 6.30
    /**
     * Show the form for creating a new resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function create()
    {
        if( Auth::user()->id == 1 ){
            $categories = Category::where('user_id', Auth::user()->id)->
            get();
            $writers = Writer::where('user_id', Auth::user()->id)->get();
            return view('article.create', compact('categories',
            'writers'));
        }
        return view('auth.login');

    }
    /**
     * Store a newly created resource in storage.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request){
        if(Auth::user()->id == 1){
          if($file = $request->file('image')){
              $name = $file->getClientOriginalName();
            //using the Article model to create posts
            $post = Article::create([
             'title' => $request->input('title'),
             'body' => $request->input('body'),
            'user_id' => Auth::user()->id,
            'category_id' => $request->input('category_id'),
            'writer_id' => $request->input('writer_id'),
            'tag' => $request->input('tag'),
            'image' => $name
        ]);
        $file->move('images/articles', $name);
   }
        if($post){
            return redirect()->route('article.show', ['post'=> $post->id])
            ->with('success', 'article created successfully');
        }
     }
     return back()->withInput()->with('errors', 'Error creating new
     article');
}
```

The edit part is different in the `article.edit` Blade page. Here is the full code:

```
//code 6.31
@extends('layouts.app')

@section('content')
<div class="container">
  <div class="card-header">Recent Tasks</div>

    <div class="row justify-content-left">
        <div class="col-md-4">
            <div class="card">
              <div class="card-body">
                Categories
                <p></p>
                Articles
                <p></p>
                Add Users
                <p></p>
                </div>
            </div>
        </div>
        <div class="col-md-8">
            <div class="card">
              <div class="card-body">
                <form enctype="multipart/form-data" method="post"
                action="{{ route('article.update', [$article->id]) }}">
                {{ csrf_field() }}

                <input type="hidden" name="_method" value="put">
                          <div class="form-group">
                             <label for="post-name">Title
                               <span class="required">*</span>
                             </label>
<input   placeholder="Enter title" id="post-title" value="{{ $article-
>title }}"
required name="title" spellcheck="false" class="form-control"/>
                              </div>
                                 @if($categories == null)
<input class="form-control" type="hidden" required name="category_id"
value="{{ $category_id }}"/>
                          </div>
                                 @endif

                                      @if($categories != null)
                                      <div class="form-group">
                                          <label for="category-
                                          content">Select Category</label>
                                          <span class="required">*</span>

                                          <select name="category_id"
                                          class="form-control" >


                                          @foreach($categories as $category)
                                                  <option
                                                  value="{{$category->id}}">
```

```
                                        {{$category->name}}
                                      </option>
                                    @endforeach
                              </select>
                          </div>
                          @endif

                <div class="form-group">
                              <label for="project-
                              content">News Content</label>
                              <span class="required">*</span>
                              <textarea placeholder="Enter
                              body"
                                        style="resize:
                                        vertical"
                                        id="post-body"
                                        required
                                        name="body"
                                        rows="10"
                                        spellcheck="false"
                                        class="form-control
                                        autosize-target text-
                                        left">
                                        {{ $article->body }}
                               </textarea>
                          </div>

                          @if($writers == null)
                          <input class="form-control"
                          type="hidden" required
                          name="writer_id"
                          value="{{ $writer_id }}"/>
                          </div>
                          @endif

                               @if($writers != null)
                               <div class="form-group">
```

```
          <label for="category-
          content">Select Writer</
          label>
          <span class="required">*</
          span>

          <select name="writer_id"
          class="form-control" >

          @foreach($writers as
          $writer)
                  <option value="{{
                  $writer->id }}">
                    {{ $writer->name
                    }}
                  </option>
                @endforeach
          </select>
      </div>
      @endif

 <div class="form-group">
    <label class="form-group">
        <input type="hidden"
        name="MAX_FILE_SIZE"
        value="3000000" />
        <input name="image"
        type="file">
       <span class="custom-file-
       control">Upload Image</span>
     </label>
</div>

<div class="form-group">
   <label for="post-name">Tag
     <span class="required">*</span>
   </label>
```

```
             <input   placeholder="Enter Tags" id="article-title" value="{{
             $article->tag }}"
             required name="tag" spellcheck="false" class="form-control"/>
                                  </div>

                          <div class="form-group">
                            <input type="submit" class="btn btn-primary"
                                   value="Submit"/>
                        </div>
         </form>

               </div>


            </div>
         </div>

    </div>
</div>
@endsection
```

The first four lines are important, as shown here:

```
//code 6.32
<form enctype="multipart/form-data" method="post"
                action="{{ route('article.update', [$article->id]) }}">
                {{ csrf_field() }}
                <input type="hidden" name="_method" value="put">
```

In the `action` part, you pass the content ID. Although the method has been mentioned as `put`, you pass the hidden `put` method. Laravel is smart enough to understand this and use `put` to edit the contents.

Since you have set the action to `article.update`, `ArticleController` contains the following logic, as shown here:

```
/**
     * Show the form for editing the specified resource.
     *
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function edit(Article $article)
    {
        //
        $article = Article::find($article->id);
        $categories = Category::all();
        $writers = Writer::all();

        return view('article.edit', compact('article', 'categories',
        'writers'));
    }

    /**
     * Update the specified resource in storage.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function update(Request $request, Article $article)
    {
        //
        if(Auth::user()->id == 1){
          if($file = $request->file('image')){

              $name = $file->getClientOriginalName();

         $post = Article::where('id', $article->id)->update([
             'title' => $request->input('title'),
             'body' => $request->input('body'),
             'user_id' => Auth::user()->id,
             'category_id' => $request->input('category_id'),
             'writer_id' => $request->input('writer_id'),
             'tag' => $request->input('tag'),
             'image' => $name
         ]);
```

```
            $file->move('images/articles', $name);
    }
    else {
      // code...
      $post = Article::where('id', $article->id)->update([
          'title' => $request->input('title'),
          'body' => $request->input('body'),
          'user_id' => Auth::user()->id,
          'category_id' => $request->input('category_id'),
          'writer_id' => $request->input('writer_id'),
          'tag' => $request->input('tag')
      ]);
    }
       if($post){
         $id = $article->id;
              return redirect()->route('article.show', compact('id'))
              ->with('success', 'article created successfully');
          }
     }
  }
```

Form Request Validation

You have seen how you can manipulate your own validation logic in the controller. However, for more complex scenarios, you can create a *form request*. These are custom request classes, and you can place your validation logic within them.

To create a form request class, use this command in your terminal:

```
//code 6.33
$ php artisan make:request StoreArticle
```

This class is generated and is placed in the `app/Http/Requests` directory. The file comes with these lines of code:

```
<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class StoreArticle extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize()
    {
        return false;
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules()
    {
        return [
            //
        ];
    }
}
```

Since this is a custom class, you need to place your logic inside the `rules()` method.

After adding more functionality to your validation logic, it looks like this:

```
//code 6.34

public function authorize()
    {
        return true;
    }
    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules()
    {
        return [
            'title' => 'required|unique:users|max:255',
            'body' => 'required',
        ];
    }
    public function messages()
    {
        return [
            'title.required' => 'Title is required!',
            'body.required' => 'Content is required!',
        ];
    }
```

Now you can flash the custom message if the validation rules fail. Besides, you have set the `authorize()` method to `return true;`. If the user is not authorized, this will display a default unauthorized page, as shown in Figure 6-7.

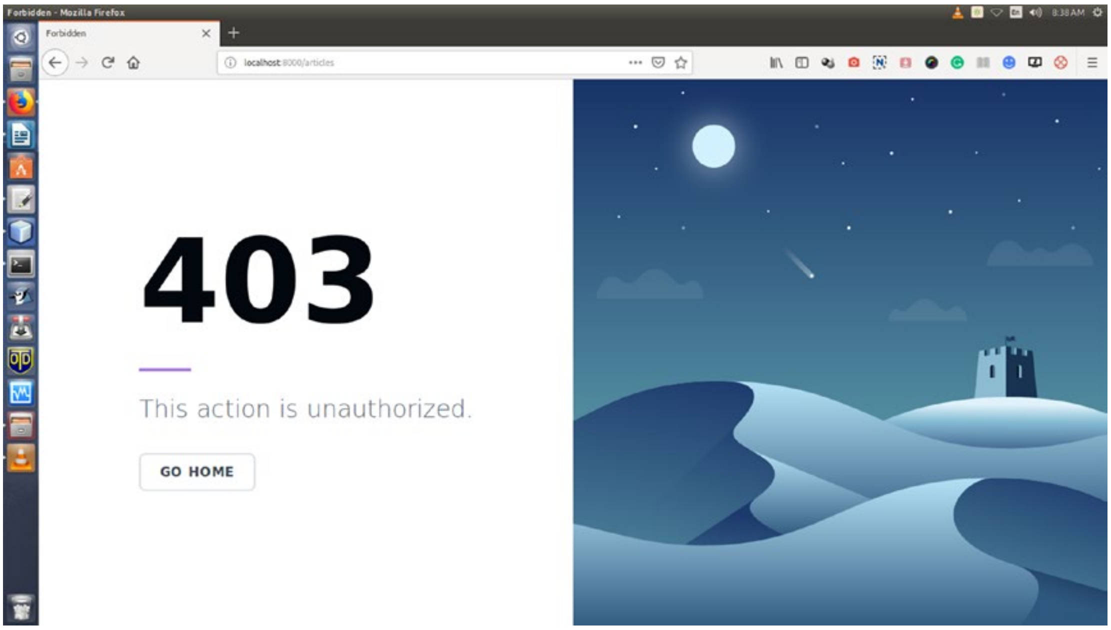


Figure 6-7. *When custom validation is unauthorized*

This happens because Laravel wants the authorization to be valid. John cannot and should not edit Jean's comments. In this case, since there is just one administrator, you can set the `is_admin` attribute to `true` for the first user (code 6.30); and if you are creating your own content, you can make it `true`.

To summarize, you can conclude that form inputs are an integral part of any Laravel application. Without forms, you cannot use CRUD or make your application dynamic. Laravel offers you many choices, and you can choose any one of them to take your application to the next level.

You have so far learned about creating, retrieving, updating, and deleting your news articles in the administrative dashboard. You have also learned to upload images and select categories and learned about validating inputs, displaying custom error outputs, and so on. However, so far you have seen only one part of authentication. You will learn about other methods in detail in Chapter 8 when building another role-based dynamic application.

CHAPTER 7

Using Tinker

Since Laravel depends heavily on command-line interfaces, you will need to know a number of helpful commands to assist you while you build your applications.

Let's first try to understand what a command-line interface is. Basically, it is a type of interactive shell that takes in single-user inputs, evaluates them, and returns the result to the user, in the form of a read-eval-print-loop (REPL) mechanism. Therefore, you can say that Tinker is a kind of REPL.

Actually, PHP has its own interactive shell, called PsySH; Justin Hileman created it, and Tinker is powered by PsySH.

Tinker helps you when you want to do some quick CRUD operations in the database records of your application through the terminal. Before Laravel 5.4, it was part of the Laravel package; now in Laravel 5.8 it extracts itself into a separate folder, although you can use it easily in your terminal.

Handling a Database Using Tinker

Writing PHP code through a command line is not easy. It is especially difficult when you want to add dummy data in your database tables and you don't have immediate access to your database.

In such cases, Tinker is your friend.

You can even update or delete table records in the database through the Tinker.

As mentioned, Tinker is a REPL powered by the PsySH (`https://github.com/bobthecow/psysh`) package. In Laravel, the use of Tinker is not limited to only database handling. Tinker allows you to interact with your entire Laravel application on the command line, including the Eloquent ORM, jobs, events, and more. To enter the Tinker environment, run the `artisan tinker` command from the Laravel codebase, as follows:

```
//code 7.19
$ php artisan tinker
```

Remember, you have to be inside the Laravel application environment for this to work. And you'll want to be working on a database that has a connection to your application because you'll want to use your Eloquent models.

Once you have entered the Tinker environment, you can use any model you have. Suppose you have a model called `Listtodo`; you can create a new `list` instance like this:

```
//code 7.20
$list = new Listtodo;
$list->name = 'First Job';
$list->description = 'Go to market';
$list->save();
```

Once you issue the `save()` command, the records are saved in your database tables.

You could have created the same thing with a single command, as shown here:

```
//code 7.21
$list = Listtodo::create(
array('name' => 'First Job',
'description' => 'Go to Market')
);
```

You can get the output of all lists with this command:

```
//code 7.22
echo Listtodo::all()->count();
```

You can get the first record, as shown here:

```
//code 7.23
$lists = Listtodo::select('name', 'description')->first();
```

You can get all lists ordered by name, as shown here:

```
//7.24
$lists = Listtodo::orderBy('name')->get();
```

Here is the command to get the list in descending order:

```
//code 7.25
$lists = Listtodo::orderBy('name', 'DESC')->get();
```

```
// order results by multiple columns
lists = Listtodo::orderBy('created_at', 'DESC')->orderBy('name', 'ASC')->
get();
```

You have many choices such as using conditionals, where you can check whether your listed job is completed.

```
//code 7.26
$lists = Listtodo::where('isComplete', '=', 1)->get();
```

Suppose you want to get five records in descending order, as shown here:

```
//code 7.27
$lists = Listtodo::take(5)->orderBy('created_at', 'desc')->get();
```

You may want to skip five records, as shown here:

```
//code 7.28
lists = Listtodo::take(5)->skip(5)->orderBy('created_at', 'desc')->get();
```

For a random output of records, these commands are useful:

```
//code 7.29
$list = Listtodo::all()->random(1);
$list = Listtodo::orderBy(DB::raw('RAND()'))->first();
```

For a quick CRUD operation, Tinker is extremely useful. Suppose you want to update a part of your existing records, as shown here:

```
//code 7.30
$list = Listtodo::find(1);
$list->name = 'Going to market and buying fish';
$list->save();
```

The effect may be achieved with this command, as shown here:

```
//code 7.31
$list = Listtodo::updateOrCreate(
array('name' => 'Going to market and buying fish'),
);
```

Deleting a record is easy. You can use either the `delete()` or `destroy()` method. The advantage of the `destroy()` method is that it takes a record ID as the parameter, as shown here:

```
//code 7.32
$list = Listtodo::find(2);
$list->delete();
Listtodo::destroy(2);
```

Using a DB facade in Tinker is often useful. In that case, you don't use Eloquent ORM; instead, you use a database facade directly. However, you get the same effect.

```
//code 7.33
$lists = DB::table('liststodos')->get();
foreach ($lists as $list) {
echo $list->name;
}
```

In this case, you need to identify the lists by ID. Here you want to find the ID number 6:

```
//code 7.34
$list = DB::table('liststodos')->find(6);
```

Here you get all the names in one go:

```
//code 7.35
$lists = DB::table('liststodos')->select('name')->get();
```

Traditional querying is also possible, as shown here:

```
//code 7.36
$lists = DB::select('SELECT * from todolists');
```

Inserting data into the database tables is also easy, as shown here:

```
//code 7.37
DB::insert('insert into todolists (name, description) values (?, ?)',
array('Second job', 'Finishing the last chapter');
```

With a single command, you can delete a record, as shown here:

```
//code 7.38
DB::delete('delete from todolists where completed = 1');
```

Want to drop a table entirely? Well, you don't have to open your MySQL wizards; you can achieve the result in your terminal, as shown here:

```
//code 7.39
$lists = DB::statement('drop table todolists');
```

Tinker is helpful when doing any database operation, not only the small and easy ones but the complex ones too.

SQLite Is a Breeze!

If you want to make a new company/project/task management application, which is also a CRUD-based application, you can make this application entirely based on a SQLite database. However, for a big and complex application, people opt for MySQL or PgSQL because each of these can handle more visitors.

SQLite may not be big enough and basically file-based and light in nature, but it can easily tackle small to medium applications with 100,000 visitors. So, you can feel free to use it for any small or medium-sized CRUD applications. Especially for Laravel, SQLite is a breeze to use because you can use Tinker to manipulate the database.

To use SQLite in Laravel, you need to change the default database setup. Two lines need to be changed. Here's the first one:

```
//Code/test/blog/config/databse.php
'default' => env('DB_CONNECTION', 'sqlite'),
```

In the second line, you need to mention the SQLite database file path, as shown here:

```
//Code/test/blog/config/databse.php
'connections' => [

        'sqlite' => [
            'driver' => 'sqlite',
            //'database' => storage_path('database.sqlite'),
            'database' => env('DB_DATABASE', database_path('/../database/
            database.sqlite')),
            'prefix' => '',

        ],
```

Suppose your local Laravel application is in the `Code/test/blog` directory. In that case, you will keep your SQLite file in the `Code/test/blog/database/` folder. Many people go for the `storage` folder. Either one of them will work.

Second, you need to change the `.env` file. In the original file that comes with Laravel, the default database is mentioned like this:

```
//Code/test/blog/.env
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=testdb
DB_USERNAME=root
DB_PASSWORD=pass
```

You must change it to this:

```
//Code/test/blog/.env
DB_CONNECTION=sqlite
DB_HOST=127.0.0.1
DB_PORT=3306
```

From now on, any database operation you do on your application will automatically be registered on the SQLite database file that you create in the `Code/test/blog/database` folder. Normally people don't create a new SQLite file in the `Code/test/blog/database` folder. They choose the `Code/test/blog/storage/databse.sqlite` file that comes with Laravel by default. In that case, you need to change the database path `Code/test/blog/config/databse.php`.

If you want to take a little break from the usual path and create a `database.sqlite` file in the `Code/test/blog/database` folder, you must go to the desired folder first, as shown here:

```
cd Code/test/blog/database
```

Next you have to use the `touch` command to create the file.

```
touch database.sqlite
```

Now you're ready to make any type of CRUD application using SQLite.

CHAPTER 8

Authentication, Authorization, and Middleware

The Internet is an open gateway, and ideally data should be able to travel on it freely. However, to be secure, this free flow of data has to be monitored and blocked sometimes. In today's world, the Web reaches into almost every part of our lives, so we need the proper security in place, and your Laravel applications are no exception.

Today, most applications need to authenticate users, at least in some areas. In other areas, your application might need to implement authorization, because authentication by itself is not enough. That's why most applications have layers of security and several different roles, such as administrators, moderators, and general members.

It usually takes time to create proper authentication and authorization classes, but not in Laravel. Implementing the functionality to authenticate users is simple. Further, it is simple to add authorization to work seamlessly with the authentication process.

To make this all possible, you need to understand the filtering process that Laravel adopts while it allows requests to enter the application. Laravel ships with several prebuilt authentication controllers. You can view them in the `App\Http\Controllers\Auth` namespace. There are controllers such as `RegisterController`, which handles new user registration; `LoginController`, which handles authentication; `ForgotPasswordController`, which manages e-mailing links for resetting passwords; and `ResetPasswordController`, which contains the logic to reset passwords. For most applications, you will not need to even tweak these controllers.

S. Sinha, *Beginning Laravel*, https://doi.org/10.1007/978-1-4842-4991-8_8

You learned earlier in the book that a single command, `php artisan make:auth`, solves the authentication problem for developers. At the same time, Laravel provides a quick way to scaffold all the routes and views you need for authentication using that single command.

This `make:auth` command creates a `HomeController` and `resources/views/layouts` directory containing a base layout for your application (although you are free to customize the layout).

Different Authentication Methods in the Company/ Project/Task Management Application

Before learning about the role-based methods of authentication in Laravel, let's take a quick look at the new application you are going to build in this chapter.

In Chapter 6, I discussed why we need a new application to learn these different methods of role-based authentication. The previous news application was managed by a single administrator. Here, in the company/project/task management application, you will have different types of users who will manage different types of resources.

For example, a project manager or moderator cannot view the administrator's dashboard. A general user cannot penetrate the moderator's dashboard. You can create a workflow like this in various ways. For example, you can use middleware, you can customize the roles through the `users` table, or you can authorize a user by applying gates and policies. In this chapter, you will see each implementation separately.

Now, for brevity, I cannot show you all the code for all implementations in this chapter, because it would add thousands of lines of code to the book. What I can do is show the basic code snippets so that you can understand the workflow. The entire application code is available with the download for the book; I suggest you download the files to connect the dots as necessary.

To get started, let's think about the `companies` resource first. In your application structure, it sits at the top, and only the administrators can add projects to that resource. Here is the `routes/web.php` code:

```
//routes/web.php
<?php

/*
|--------------------------------------------------------------------------
| Web Routes
|--------------------------------------------------------------------------
|
| Here is where you can register web routes for your application. These
| routes are loaded by the RouteServiceProvider within a group which
| contains the "web" middleware group. Now create something great!
|
*/
//use App\Http\Middleware\CheckRole;

Route::group(['middleware' => ['web', 'auth']], function(){
  Route::get('/adminonly', function () {
    if(Auth::user()->admin == 0){
      return view('restrict');
    }else{
      $users['users'] = \App\User::all();
      return view('adminonly', $users);
    }
  });
});

Route::get('/admin', function () {
  if (Gate::allows('admin-only', Auth::user())) {
        // The current user can view this page
        return view('admin');
    }
    else{
      return view('restrict');
    }
});
```

```php
Route::get('/mod', function () {
  if (Gate::allows('mod-only', Auth::user())) {
        // The current user can view this page
        return view('mod');
    }
    else{
      return view('restrict');
    }
});

Auth::routes();

Route::resource('home', 'HomeController');

Route::resource('users', 'UserController');

Route::resource('companies', 'CompanyController');

Route::resource('companies', 'CompanyController');

Route::resource('projects', 'ProjectController');

Route::resource('roles', 'RoleController');
Route::resource('tasks', 'TaskController');

Route::resource('comments', 'CommentController');

Route::resource('articles', 'ArticleController');
Route::get('/users/{id}/articles', 'ArticleController@articles');
Route::resource('reviews', 'ReviewController');
Route::get('/users/{id}/reviews', 'ReviewController@reviews');

Route::get('companies/destroy/{id}', ['as' => 'companies.get.destroy',
        'uses' => 'CompanyController@getDestroy']);
```

In the `routes/web.php` code, you use middleware and role-based authentication; specifically, you'll find that code in this part:

```php
Route::group(['middleware' => ['web', 'auth']], function(){
  Route::get('/adminonly', function () {
    if(Auth::user()->admin == 0){
      return view('restrict');
    }else{
      $users['users'] = \App\User::all();
      return view('adminonly', $users);
    }
  });
});
```

You will learn how this works in a minute.

In the second part, you are using gates and policies, like this:

```
Route::get('/admin', function () {
  if (Gate::allows('admin-only', Auth::user())) {
        // The current user can view this page
        return view('admin');
    }
    else{
      return view('restrict');
    }
});

Route::get('/mod', function () {
  if (Gate::allows('mod-only', Auth::user())) {
        // The current user can view this page
        return view('mod');
    }
    else{
      return view('restrict');
    }
});
```

This separates the administrator from the moderators, giving them the freedom to work on their own pages. I will discuss them in detail in this chapter.

Next take a look at the `CompanyController.php` file, as shown here:

```
//app/Http/Controllers/CompanyController.php
<?php

namespace App\Http\Controllers;
use App\User;
use App\Company;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;

class CompanyController extends Controller
{
    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        if( Auth::check() ){
            $companies = Company::where('user_id', Auth::user()->id)->get();

            if(Auth::user()->role_id == 1){
                return view('companies.index', ['companies'=> $companies]);
            }
        }
        return view('auth.login');
    }

    /**
     * Show the form for creating a new resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function create()
    {
      if( Auth::check() ){
        if(Auth::user()->role_id == 1){
                return view('companies.create');
        }
      }
        return view('auth.login');

    }
```

```
/**
 * Store a newly created resource in storage.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return \Illuminate\Http\Response
 */
public function store(Request $request)
{
    if(Auth::check()){
        $company = Company::create([
            'name' => $request->input('name'),
            'description' => $request->input('description'),
            'user_id' => Auth::user()->id
        ]);

        if($company){
            return redirect()->route('companies.show', ['company'=>
            $company->id])
                    ->with('success' , 'Company created successfully');
        }

    }

        return back()->withInput()->with('errors', 'Error creating new
        company');

}

/**
 * Display the specified resource.
 *
 * @param  \App\Company  $company
 * @return \Illuminate\Http\Response
 */
```

```
    public function show(Company $company)
    {
      if( Auth::check() ){
        if(Auth::user()->role_id == 1){
            $company = Company::find($company->id);
            return view('companies.show', ['company' => $company]);
        }
      }
        return view('auth.login');

    }

    /**
     * Show the form for editing the specified resource.
     *
     * @param  \App\Company  $company
     * @return \Illuminate\Http\Response
     */
    public function edit(Company $company)
    {
      if( Auth::check() ){
        if(Auth::user()->role_id == 1){
        $company = Company::find($company->id);

        return view('companies.edit', ['company' => $company]);
      }
    }
    }

    /**
     * Update the specified resource in storage.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \App\Company  $company
     * @return \Illuminate\Http\Response
     */
    public function update(Request $request, Company $company)
    {
        $updateCompany = Company::where('id', $company->id)->update(
                [
                    'name'=> $request->input('name'),
                    'description'=> $request->input('description')
                ]
        );

      if($updateCompany){
          return redirect()->route('companies.show', ['company'=>
          $company->id])
          ->with('success' , 'Company updated successfully');
      }
      //redirect
      return back()->withInput();
    }

    /**
     * Remove the specified resource from storage.
     *
     * @param  \App\Company  $company
     * @return \Illuminate\Http\Response
     */
    public function destroy(Company $company)
    {

    }
    public function getDestroy($id)
    {
        $company = Company::findOrFail($id);
        if($company->destroy($id)){
            return redirect()->route('companies.index')->with('success' ,
            'Company deleted successfully');
        }

    }
}
```

You learn about some of these concepts when you created the news application. For example, you have already learned about the model relations, and the `Company` model follows the same rules.

```
//app/Company.php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Company extends Model
{
  /**
   * The attributes that are mass assignable.
   *
   * @var array
   */
  protected $fillable = [
      'name', 'description', 'user_id'
  ];

  public function user() {
      return $this->belongsTo('App\User');
  }

  public function project() {
      return $this->belongsTo('App\Project');
  }

  public function projects() {
      return $this->belongsToMany('App\Project');
  }

  public function reviews() {
      return $this->belongsToMany('App\Review');
  }
  public function comments()
  {
      return $this->morphMany('App\Comment', 'commentable');
  }
}
```

In this new application, the database seeder code is a little different from the previous news application. To give you an idea, let's take a look at both `UserFactory.php` and `DatabaseSeeder.php`.

First, here's the `database/factories/UserFactory.php` code:

```
// database/factories/UserFactory.php
<?php

use Faker\Generator as Faker;

/*
|--------------------------------------------------------------------------
| Model Factories
|--------------------------------------------------------------------------
|
| This directory should contain each of the model factory definitions for
| your application. Factories provide a convenient way to generate new
| model instances for testing / seeding your application's database.
|
*/

$factory->define(App\User::class, function (Faker $faker) {
    return [
        'name' => $faker->name,
        'email' => $faker->unique()->safeEmail,
        'password' => '$2y$10$TKh8H1.PfQx37YgCzwiKb.
         KjNyWgaHb9cbcoQgdIVFlYg7B77UdFm', // secret
        'remember_token' => str_random(10),
    ];
});
```

```
$factory->define(App\Company::class, function (Faker $faker) {
    return [
        'user_id' => 21,
        'name' => $faker->sentence,
        'description' => $faker->paragraph(random_int(3, 5))
    ];
});

$factory->define(App\Project::class, function (Faker $faker) {
    return [
      'name' => $faker->sentence,
      'description' => $faker->paragraph(random_int(3, 5)),
      'company_id' => App\Company::all()->random()->id,
      'user_id' => 21,
      'days' => $faker->biasedNumberBetween($min = 1, $max = 20, $function
      = 'sqrt')
    ];
});

$factory->define(App\Role::class, function (Faker $faker) {
    return [
        'name' => $faker->word
    ];
});

$factory->define(App\Task::class, function (Faker $faker) {
    return [
        'name' => $faker->word,
        //'user_id' => App\User::all()->random()->id,
        'user_id' => 21,
        'project_id' => App\Project::all()->random()->id,
        'company_id' => App\Company::all()->random()->id,
        'days' => $faker->biasedNumberBetween($min = 1, $max = 20,
        $function = 'sqrt')
    ];
});
```

```
$factory->define(App\Profile::class, function (Faker $faker) {
    return [
        'user_id' => App\User::all()->random()->id,
        'city' => $faker->city,
        'about' => $faker->paragraph(random_int(3, 5))
    ];
});

$factory->define(App\Country::class, function (Faker $faker) {
    return [
      'name' => $faker->country
    ];
});

$factory->define(App\Comment::class, function (Faker $faker) {

    return [
      'user_id' => $faker->biasedNumberBetween($min = 1, $max = 10,
      $function = 'sqrt'),
        'body' => $faker->paragraph(random_int(3, 5)),
        'commentable_id' => $faker->randomDigit,
        'commentable_type' => function(){
          $input = ['App\Task', 'App\Profile', 'App\Article', 'App\
          Review'];
          $model = $input[mt_rand(0, count($input) - 1)];
          return $model;
        }
    ];
});

$factory->define(App\Article::class, function (Faker $faker) {
    return [
        'user_id' => App\User::all()->random()->id,
        'title' => $faker->sentence,
        'body' => $faker->paragraph(random_int(3, 5))
    ];
});
```

```
$factory->define(App\Tag::class, function (Faker $faker) {
    return [
        'tag' => $faker->word
    ];
});

$factory->define(App\Review::class, function (Faker $faker) {
    return [
        'user_id' => App\User::all()->random()->id,
        'company_id' => App\Company::all()->random()->id,
        'title' => $faker->sentence,
        'body' => $faker->paragraph(random_int(3, 5))
    ];
});
```

Here is the seeder code:

```
//database/seeds/DatabaseSeeder.php

<?php

use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     *
     * @return void
     */
    public function run()
    {
      // $this->call(UsersTableSeeder::class);
      factory(App\User::class, 20)->create()->each(function($user){
          $user->profile()->save(factory(App\Profile::class)->make());
      });

        factory(App\Company::class, 10)->create()->each(function($company){
          $ids = range(1, 50);
```

```
  shuffle($ids);
  $sliced = array_slice($ids, 1, 20);
  $company->projects()->attach($sliced);
});

factory(App\Project::class, 30)->create()->each(function($project){
  $ids = range(1, 50);
  shuffle($ids);
  $sliced = array_slice($ids, 1, 20);
  $project->users()->attach($sliced);
});

factory(App\Role::class, 4)->create()->each(function($role){
  $ids = range(1, 5);
  shuffle($ids);
  $sliced = array_slice($ids, 1, 20);
  $role->users()->attach($sliced);
});

factory(App\Task::class, 100)->create()->each(function($task){
  $ids = range(1, 50);
  shuffle($ids);
  $sliced = array_slice($ids, 1, 20);
  $task->users()->attach($sliced);
});

factory(App\Country::class, 30)->create();

factory(App\Comment::class, 60)->create();

factory(App\Article::class, 50)->create()->each(function($article){
  $ids = range(1, 50);
  shuffle($ids);
  $sliced = array_slice($ids, 1, 20);
  $article->tags()->attach($sliced);
});

factory(App\Tag::class, 20)->create();
```

```
        factory(App\Review::class, 50)->create()->each(function($review){
          $ids = range(1, 50);
          shuffle($ids);
          $sliced = array_slice($ids, 1, 20);
          $review->tags()->attach($sliced);
        });
    }
```

Now you will proceed to learn about various authentication methods in the next sections. What you learned while building the news application will come in handy with this new application. In fact, if you study the code snippets I have shared in this chapter already, you will find they have many things in common.

How Auth Controller Works and What Auth Middleware Is

A proper authentication and authorization process should go through the filtering examinations first, it filters users along with other credentials. If the filtering examination passes, only then can authenticated users enter your application. Laravel introduces the concept of middleware in between filtering processes so that the proper filtering takes place before anything starts. You can think of middleware as a series of layers that HTTP requests must pass through before they actually hit your application. The more advanced an application becomes, the more layers that can examine the requests in different stages, and if a filtering test fails, the request is rejected entirely.

More simply, the middleware mechanism verifies whether the user is authenticated. If the user is not authenticated, the middleware sends the user back to the login page. If the middleware is happy with the user's authentication, it allows the request to proceed further into the application.

There are also other tasks that middleware has been assigned. For example, the logging middleware might log all incoming requests to your application. Since I will discuss the authentication and authorization processes in detail, you will look at the middleware that is responsible for these particular tasks later in this chapter.

In this section, you are interested in the middleware that handles authentication and CSRF protection. All of these middleware components are located in the `app/Http/Middleware` directory.

Creating middleware is easy. Open your terminal and type the following:

```
//code 8.1
$ php artisan make:middleware CheckRole
Middleware created successfully.
```

The artisan command creates your middleware, called CheckRole. To verify, let's go to the app/Http/Middleware directory and see whether it has been created. Yes, it has. The code generated at the time of creation looks like this:

```
//code 8.2
// app/Http/Middleware/CheckRole.php
<?php
namespace App\Http\Middleware;
use Closure;
class CheckRole
{
    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        return $next($request);
    }
}
```

As you can see, the namespace points to the directory structure mentioned earlier. Another interesting thing is that the method handle() passes two arguments: one is $request, and the second one is the Closure object $next.

This means you need to define the method actions in your route where you will request a URI (like /adminonly) and return a view using the closure (the anonymous function).

You need to use this middleware in a way so that only the administrator can go to the URI /adminonly; no one else, such as moderators, editors, and other members, should be able to access it.

To make this happen, you have to organize three files, listed here:

- app/Http/Middleware/CheckRole.php
- app/Http/Kernel.php
- routes/web.php

In app/Http/Middleware/CheckRole.php, you have to add some logic first so that if in the users table the admin property is set to 0, the users will be redirected to the "restricted" page. Otherwise, the next requests will follow one after the other. Therefore, your app/Http/Middleware/CheckRole.php code changes to this:

```
//code 8.3
//'app/Http/Middleware/CheckRole.php'
    public function handle($request, Closure $next)
    {
      if(auth()->check() && $request->user()->admin == 0){
        return redirect()->guest('home');
      }
      return $next($request);
    }
}
```

The home page is under the auth middleware, and it has been defined in the resourceful HomeController.php, so it actually takes the guest to the login page. So far, you have seen how you can successfully build a news application with the help of model relations. You have also seen different categories, articles, the relationships with the users, and so on. However, that application was purely based on one administrator, and you did not implement the concepts of roles there. But now you want an administrator dashboard, where the administrator can log in and create, retrieve, and update the records successfully.

Specifically, in the company/project/task management application, you will see how different roles handle different segments of the application.

In this application, a company administrator will act as a super-admin, who has all the privileges to create, retrieve, update, or delete any resource. But a project manager (also called a *moderator* or *editor*) cannot do that. The moderator's role is limited to the projects and tasks only. A general user can only write articles, add some comments, and do things like that.

The goal is to understand the concepts so that you can implement these features in any dynamic application in the future. You want to make sure that the moderator, editor, and general users will also not be able to view the administrator dashboard. You can facilitate this process in your route. Add this piece of code in your `routes/web.php` file:

```
//code 8.4
//'routes/web.php'
Route::group(['middleware' => ['web', 'auth']], function(){
  Route::get('/adminonly', function () {
    if(Auth::user()->admin == 0){
      return view('restrict');
    }else{
      $users['users'] = \App\User::all();
      return view('adminonly', $users);
    }
  });
});
```

As you see, you create `middleware` first. The code clearly mentions that if the user is not `admin`, the application should take the user to the `home` page; otherwise, listen to the next request.

What will be the next request? That is defined in the previous code.

First, you add a request and through the closure again add your main request and the closure. Since in your first request you mention the `middleware` options, you need to add that functionality to the `app/Http/Kernel.php` file. Before checking that, let's check your main request objects where you state that if the user's `admin` property is set to 0, the user must be redirected to the "restricted" page. Otherwise, the user is welcome to the `/adminonly` URI where it returns a `view` Blade template page called `adminonly.blade.php` in the `resources/views` directory. At the same time, you have sent all users' data there along with the editing facilities.

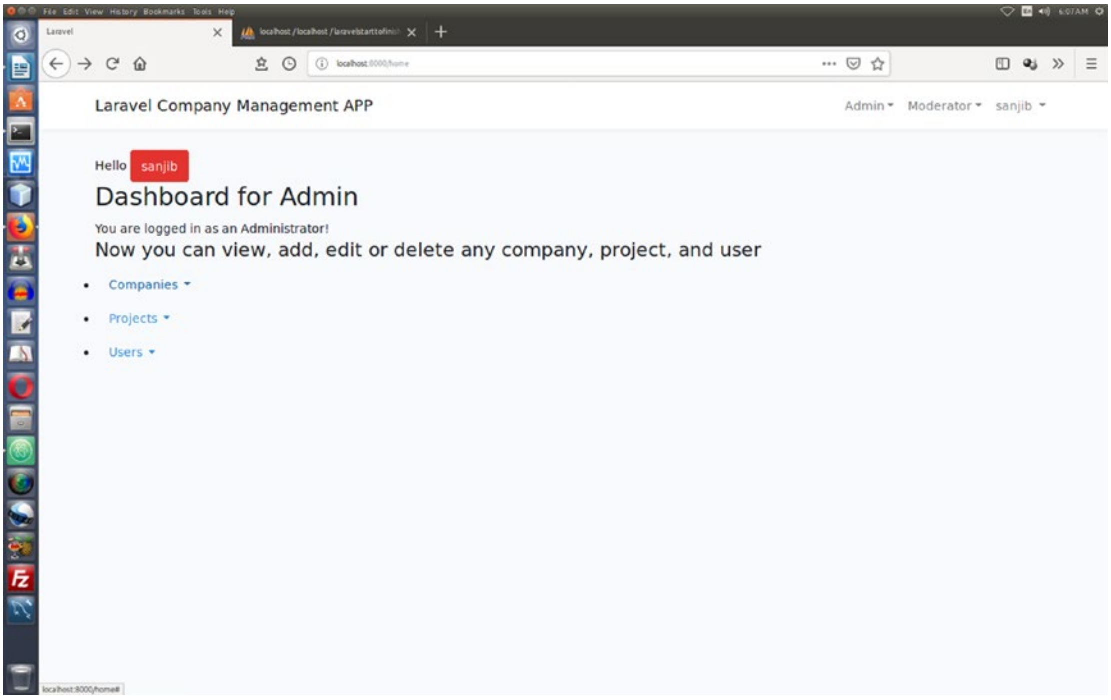
```
//code 8.5
// resources/views/adminonly.blade.php
@extends('layouts.app')
```

```
@section('content')
<div class="container">
    <div class="row">
        <div class="col-md-12 col-md-offset-2">
            <div class="panel panel-default">
                <div class="panel-heading">
                </div>
                <div class="panel-body">
                    @if (session('status'))
                        <div class="alert alert-success">
                            {{ session('status') }}
                        </div>
                    @endif
<h1 class="blog-post">THIS IS ADMIN PAGE</h1>
<h1 class="blog-post">ADMIN CAN ALSO DO</h1>
<h1 class="blog-post">SOMETHING ELSE HERE</h1>
<a href="/home">HOME</a>
<h1 class="blog-post">PLEASE VISIT ABOVE HOME LINK FOR FURTHER EDITING</h1>
<ul class="list-group">
     @foreach ($users as $user)
     <li class="list-group-item"><h2 class="blog-post-title">
             <a href="/users/{{ $user->id }}">{{ $user->name }}</a>
         </h2>
     </li><li><a href="/users/{{ $user->id }}/edit">Edit</a></li>
     @endforeach
</ul></div></div></div></div></div>
@endsection
```

This code basically gives you the output of all users with the editing facilities available to the administrator. You have, at the same time, worked on the `app/Http/Kernel.php` file and have an additional line here:

```
//code 8.6
//'app/Http/Kernel.php'
    protected $middleware = [
        \App\Http\Middleware\CheckForMaintenanceMode::class,
        \Illuminate\Foundation\Http\Middleware\ValidatePostSize::class,
        \App\Http\Middleware\TrimStrings::class,
        \Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull::
        class,
        \App\Http\Middleware\TrustProxies::class,
        \App\Http\Middleware\CheckRole::class
    ];
```

See that last line? You add the `CheckRole` class to the global HTTP middleware stack. The advantage of this middleware is that it runs during every request to your application. Altogether, you have successfully tied three files together that are needed for your middleware to work for the administrator. Figure 8-1 shows where the administrator logs in.



Hello sanjib

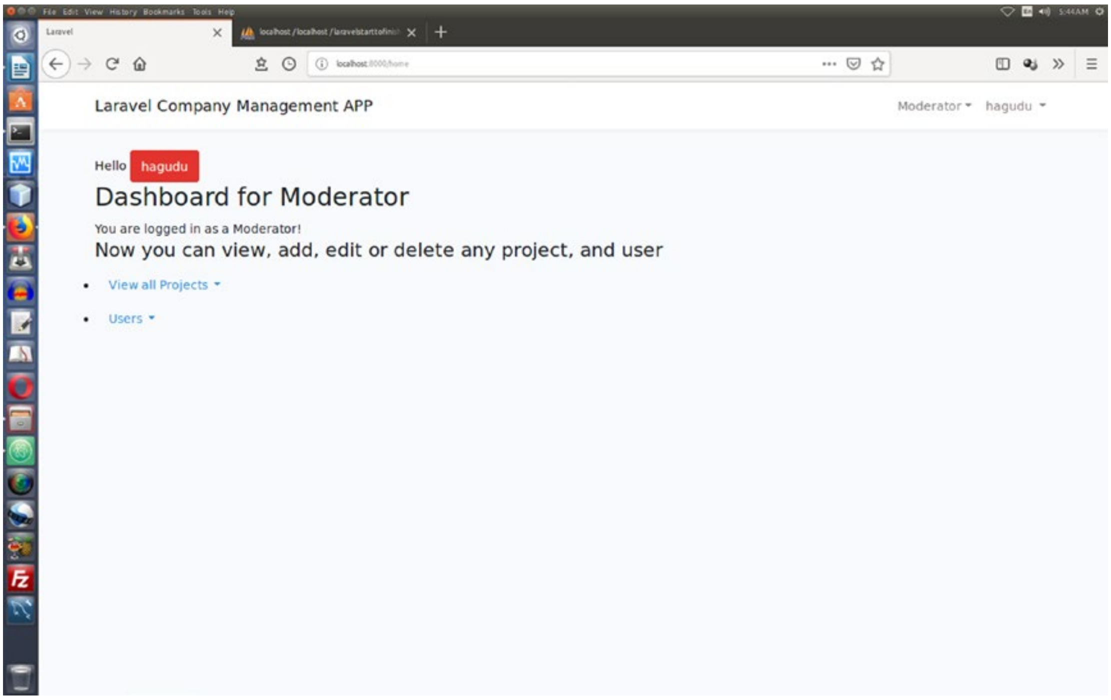
Dashboard for Admin

You are logged in as an Administrator!

Now you can view, add, edit or delete any company, project, and user

Figure 8-1. Administrator dashboard through middleware

If the moderator wants to log in, the moderator is redirected to the “restricted” page, as shown in Figure 8-2.



Hello hagudu

Dashboard for Moderator

You are logged in as a Moderator!

Now you can view, add, edit or delete any project, and user

Figure 8-2. *Administrator dashboard refusing to display when the user is a moderator*

Middleware has taught us one thing for certain: authentication plays a vital role in this filtering process. Beside authentication services, Laravel provides a simple way to authorize user actions. There are two primary ways to authorize users: gates and policies. They act like this: you need a certain policy for a certain gate. If the policy is a controller, you may think of the gate as your associated route.

In the next sections, you will see how to build an authorization process. You can build the authorization process using other ways too such as the `Role` model; it is not mandatory that you have to force the authorization process on an application only through gates and policies. You will learn about them in the next sections.

Middleware, Authentication, and Authorization in One Place

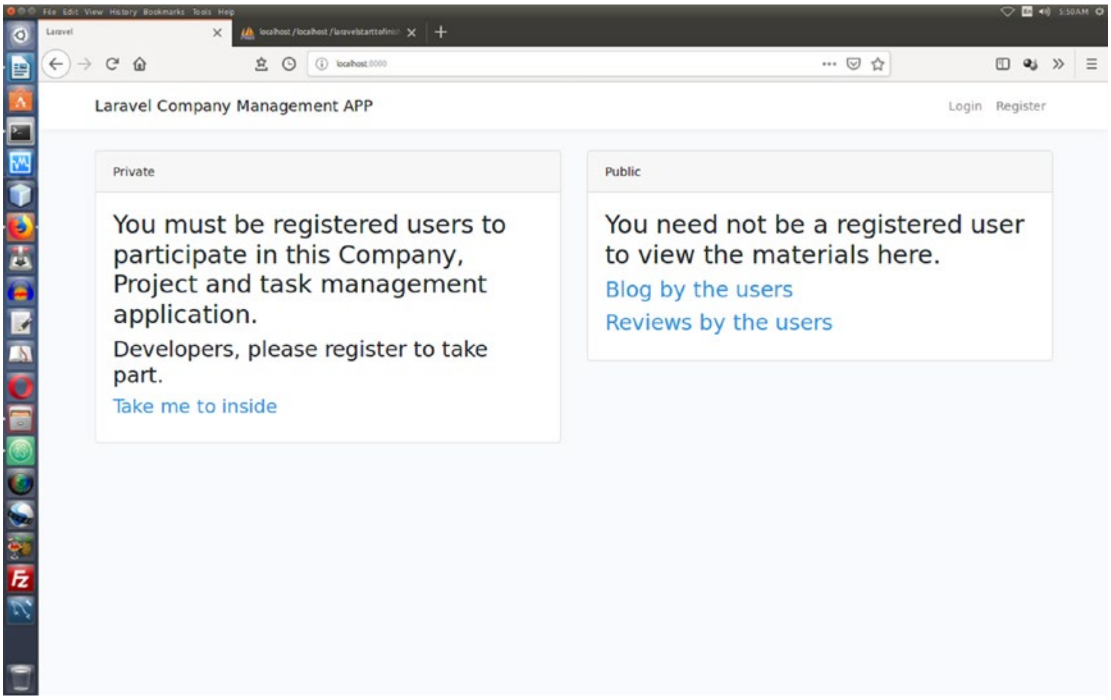
Let’s first see the routes/web.php code so that you can understand how you came to these pages. You will get the full code of the company/project/task management application in the source code section. I am going to share only the code snippets that you need for the authentication and authorization services here.

```
//code 8.7
//routes/web.php
Route::get('/', function () {
    return view('welcome');
});
/*
Route::get('/test', function () {
        //
        return view('test');
})->middleware(CheckRole::class);
*/
Route::group(['middleware' => ['web', 'auth']], function(){
  Route::get('/adminonly', function () {
    if(Auth::user()->admin == 0){
      return view('restrict');
    }else{
      $users['users'] = \App\User::all();
      return view('adminonly', $users);
    }
  });
});

Route::get('/admin', function () {
  if (Gate::allows('admin-only', Auth::user())) {
        // The current user can view this page
        return view('admin');
    }
    else{
      return view('restrict');
    }
});

Route::get('/mod', function () {
  if (Gate::allows('mod-only', Auth::user())) {
        // The current user can view this page
        return view('mod');
    }
    else{
      return view('restrict');
    }
});

Auth::routes();

Route::resource('home', 'HomeController');

Route::resource('users', 'UserController');

Route::resource('companies', 'CompanyController');

Route::resource('companies', 'CompanyController');

Route::resource('projects', 'ProjectController');

Route::resource('roles', 'RoleController');
Route::resource('tasks', 'TaskController');

Route::resource('comments', 'CommentController');

Route::resource('articles', 'ArticleController');
Route::get('/users/{id}/articles', 'ArticleController@articles');
Route::resource('reviews', 'ReviewController');
Route::get('/users/{id}/reviews', 'ReviewController@reviews');

Route::get('companies/destroy/{id}', ['as' => 'companies.get.destroy',
        'uses' => 'CompanyController@getDestroy']);
```

The first few lines are important, as they tell you about the `welcome` page and how the login mechanism works.

```
Route::get('/', function () {
    return view('welcome');
});

Auth::routes();
Route::resource('home', 'HomeController');
```

You will look at the home page code along with the `HomeController` to understand the logic of authentication. But before that, I need to clear up a few things. First, the welcome page is not under authentication, so anyone can view it, and the welcome page has two sections: public and private. The private sections are meant for the members, and the public sections are open to the guests. Figure 8-3 shows the welcome page.



Laravel Company Management APP

Private

You must be registered users to participate in this Company, Project and task management application.

Developers, please register to take part.

Take me to inside

Public

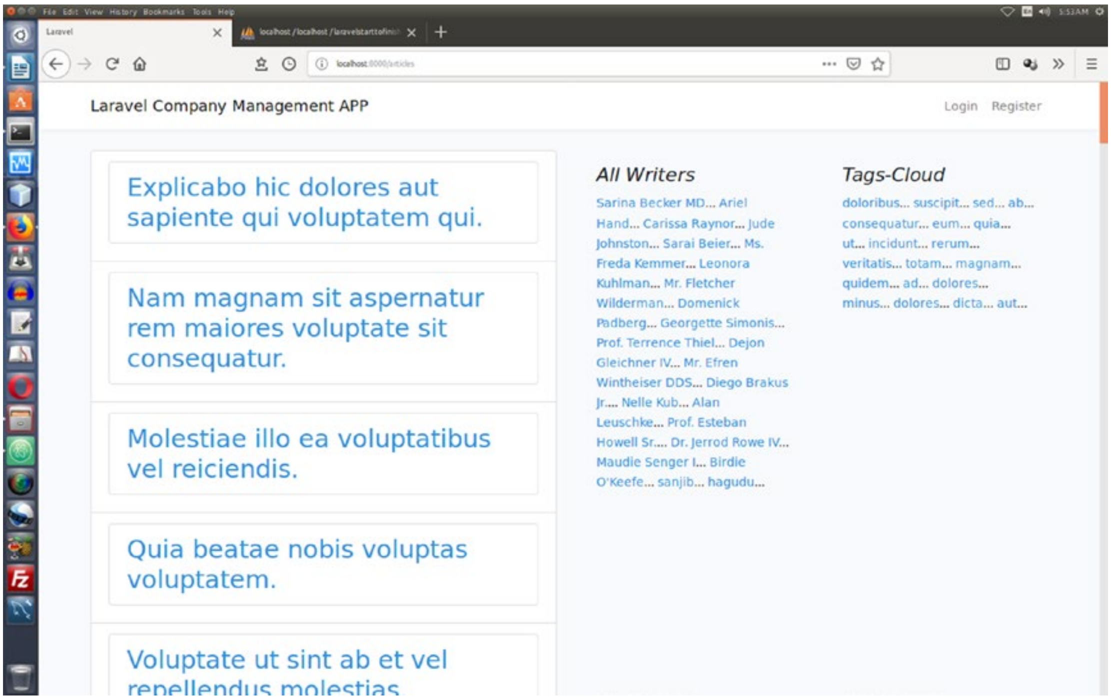
You need not be a registered user to view the materials here.

Blog by the users

Reviews by the users

Figure 8-3. *Welcome page of company management application for public viewing*

The guests can read the blogs and reviews posted by the registered users; however, some parts inside the blog section are covered by authentication, and the same is true for the reviews section. Let's click the blog link to see all the blogs first; see Figure 8-4.



Laravel Company Management APP

Login Register

Explicabo hic dolores aut sapiente qui voluptatem qui.

Nam magnam sit aspernatur rem maiores voluptate sit consequatur.

Molestiae illo ea voluptatibus vel reiciendis.

Quia beatae nobis voluptas voluptatem.

Voluptate ut sint ab et vel repellendus molestias

All Writers

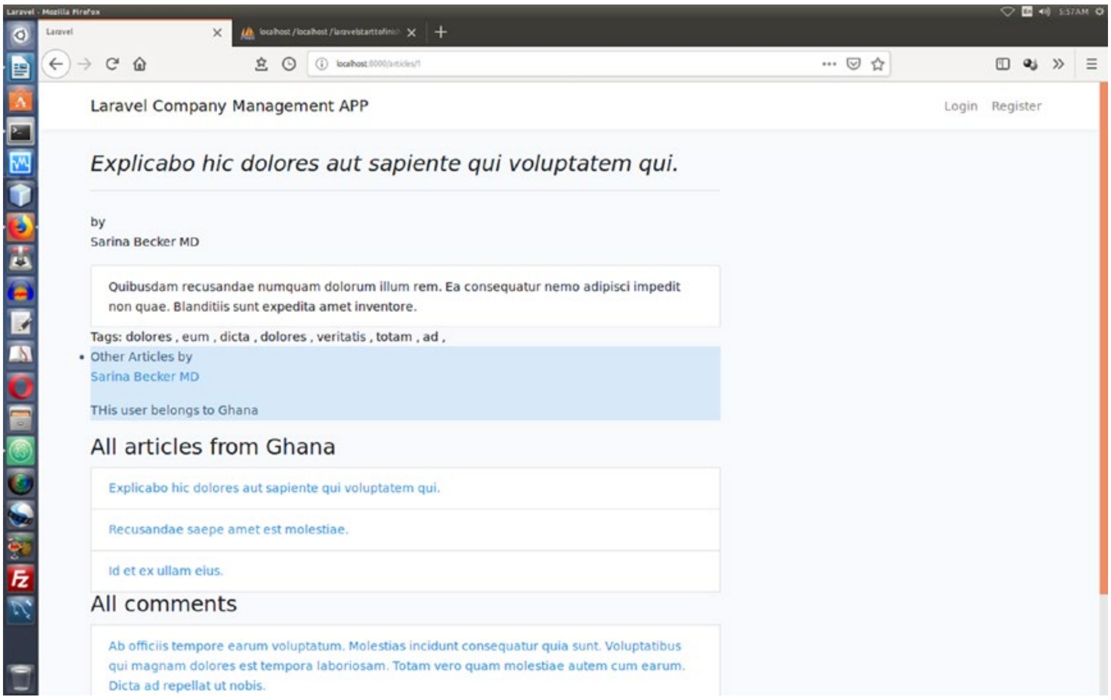
Sarina Becker MD... Ariel Hand... Carissa Raynor... Jude Johnston... Sarai Beier... Ms. Freda Kemmer... Leonora Kuhlman... Mr. Fletcher Wilderman... Domenick Padberg... Georgette Simonis... Prof. Terrence Thiel... Dejon Gleichner IV... Mr. Efren Wintheiser DDS... Diego Brakus Jr.... Nelle Kub... Alan Leuschke... Prof. Esteban Howell Sr.... Dr. Jerrod Rowe IV... Maudie Senger I... Birdie O'Keefe... sanjib... hagudu...

Tags-Cloud

doloribus... suscipit... sed... ab... consequatur... eum... quia... ut... incidunt... rerum... veritatis... totam... magnam... quidem... ad... dolores... minus... dolores... dicta... aut...

Figure 8-4. *The blogs by the users along with their names and tags*

Here the URI is `http://localhost/articles`. Now let's click the first article and see how it looks; see Figure 8-5.


Laravel Company Management APP

Login Register

Explicabo hic dolores aut sapiente qui voluptatem qui.

by
Sarina Becker MD

Quibusdam recusandae numquam dolorum illum rem. Ea consequatur nemo adipisci impedit non quae. Blanditiis sunt expedita amet inventore.

Tags: dolores , eum , dicta , dolores , veritatis , totam , ad ,

- Other Articles by
Sarina Becker MD

THis user belongs to Ghana

All articles from Ghana

Explicabo hic dolores aut sapiente qui voluptatem qui.

Recusandae saepe amet est molestiae.

Id et ex ullam eius.

All comments

Ab officiis tempore earum voluptatum. Molestias incidunt consequatur quia sunt. Voluptatibus qui magnam dolores est tempora laboriosam. Totam vero quam molestiae autem cum earum. Dicta ad repellat ut nobis.

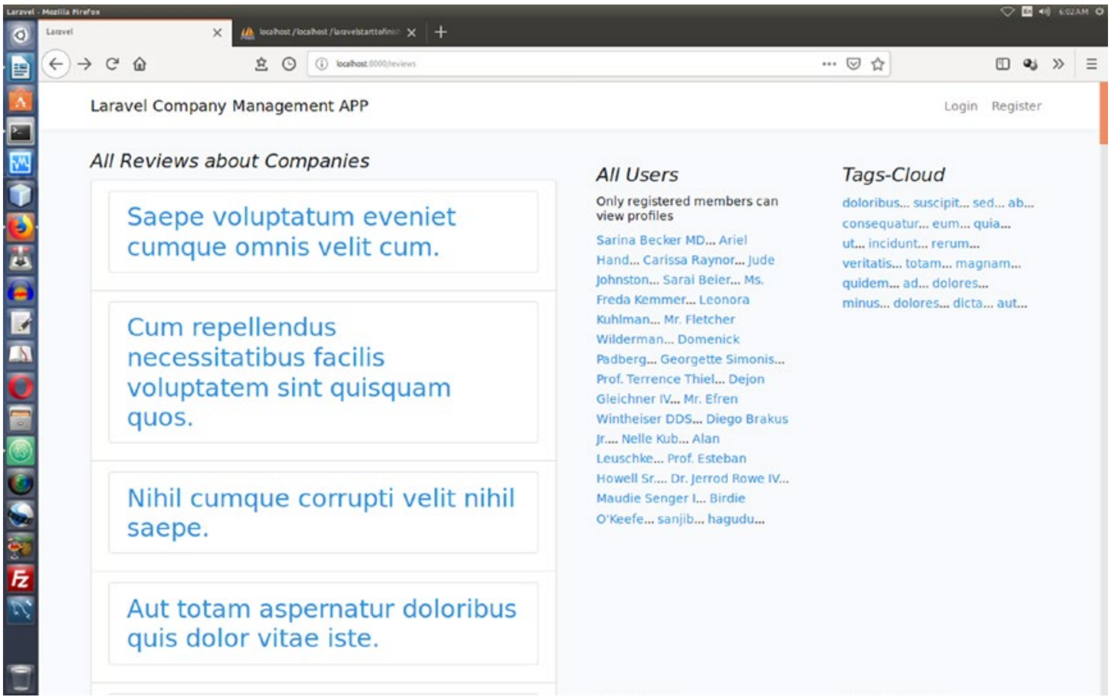
Figure 8-5. *The first article*

The URI is quite simple to follow: `http://localhost/articles/1`.

However, this page has many layers; for example, you can also read other articles by the user. Since this user is from Ghana (the faker object has chosen this country for this user), you can also view other articles written by members from the same country. You can also view all the comments posted on this page against this article.

On the welcome page, the link to the reviews section works the same way.

On the welcome page, when you click the reviews link, it takes you to the page shown in Figure 8-6.


Laravel Company Management APP

Login Register

All Reviews about Companies

Saepe voluptatum eveniet cumque omnis velit cum.

Cum repellendus necessitatibus facilis voluptatem sint quisquam quos.

Nihil cumque corrupti velit nihil saepe.

Aut totam aspernatur doloribus quis dolor vitae iste.

All Users

Only registered members can view profiles

Sarina Becker MD... Ariel Hand... Carissa Raynor... Jude Johnston... Sarai Beier... Ms. Freda Kemmer... Leonora Kuhlman... Mr. Fletcher Wilderman... Domenick Padberg... Georgette Simonis... Prof. Terrence Thiel... Dejon Gleichner IV... Mr. Efren Wintheiser DDS... Diego Brakus Jr.... Nelle Kub... Alan Leuschke... Prof. Esteban Howell Sr.... Dr. Jerrod Rowe IV... Maudie Senger I... Birdie O'Keefe... sanjib... hagudu...

Tags-Cloud

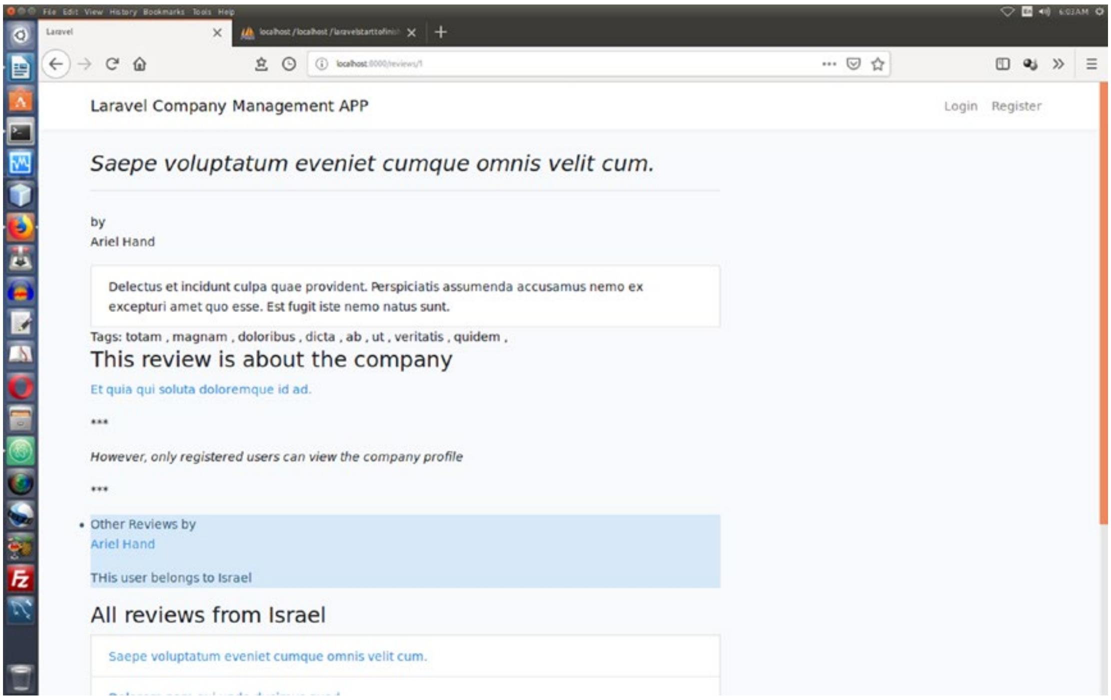
doloribus... suscipit... sed... ab... consequatur... eum... quia... ut... incidunt... rerum... veritatis... totam... magnam... quidem... ad... dolores... minus... dolores... dicta... aut...

Figure 8-6. *The reviews page*

The functioning of this reviews page is almost same as the articles page except that the content is different. Let's click the first review and see what you can find inside.

The main difference is any review says something about a company, so the model relationship changes in the business layer.

A review is related to a company, and one article is related to a single user. In both cases, this application allows you to read the articles and the reviews but never allows you to read the information about the companies or the user. See Figure 8-7.


Laravel Company Management APP

Login Register

Saepe voluptatum eveniet cumque omnis velit cum.

by
Ariel Hand

Delectus et incidunt culpa quae provident. Perspiciatis assumenda accusamus nemo ex excepturi amet quo esse. Est fugit iste nemo natus sunt.

Tags: totam , magnam , doloribus , dicta , ab , ut , veritatis , quidem ,

This review is about the company

Et quia qui soluta doloremque id ad.

***

However, only registered users can view the company profile

***

- Other Reviews by
  Ariel Hand

  THis user belongs to Israel

All reviews from Israel

Saepe voluptatum eveniet cumque omnis velit cum.

Figure 8-7. *The first review*

Here the URI is simple: `http://localhost/reviews/1`. This page also lists many other things such as the company name and the link for this review, other reviews by the same user, the country name the user belongs to, and all the reviews from that country.

The Company App's Model-View-Controller

Let's go back to the article section again and try to understand the workflow between the model, view, and controller.

Here is the code of `ArticleController`, the `Article` model, and all the view page of the articles:

```
//code 8.9
//app/HTTP/Controllers/ArticleController.php
<?php

namespace App\Http\Controllers;

use App\Article;
use App\Country;
use App\User;
use App\Tag;
use Illuminate\Http\Request;

class ArticleController extends Controller
{
    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
      $articles = Article::all();
      //$articles = Article::where('active', 1)->orderBy('title', 'desc')-
      >take(10)->get();
      $users = User::all();
      $tags = Tag::all();
      return view('articles.index', compact('articles', 'users', 'tags'));
    }

    /**
     * Show the form for creating a new resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function create()
    {
        //
    }

    /**
     * Store a newly created resource in storage.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
```

```
public function store(Request $request)
{
    //
}

/**
 * Display the specified resource.
 *
 * @param  int  $id
 * @return \Illuminate\Http\Response
 */
public function show(Article $article)
{
  $tags = Article::find($article->id)->tags;
  $article = Article::find($article->id);
  $comments = $article->comments;
  $user = User::find($article->user_id);
  $country = Country::where('id', $user->country_id)->get()->first();

  return view('articles.show', compact('tags','article',
  'country', 'comments', 'user'));
}

/**
 * Display the specified resource.
 *
 * @param  \App\Article  $article
 * @return \Illuminate\Http\Response
 */
public function articles($id)
{
  $user = User::find($id);

  return view('articles.articles', compact('user'));
}
```

```
    /**
     * Show the form for editing the specified resource.
     *
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function edit($id)
    {
        //
    }

    /**
     * Update the specified resource in storage.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function update(Request $request, $id)
    {
        //
    }

    /**
     * Remove the specified resource from storage.
     *
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function destroy($id)
    {
        //
    }
}
```

As you can see, I have left a few methods blank, but you will use them in near future for inserting and updating your application. Currently, you are concerned about only three methods: `index()`, `show()`, and `articles()`.

```
    //index method
public function index()
    {
      $articles = Article::all();
      $users = User::all();
      $tags = Tag::all();
      return view('articles.index', compact('articles', 'users', 'tags'));
    }
    //show method
public function show(Article $article)
    {
      $tags = Article::find($article->id)->tags;
      $article = Article::find($article->id);
      $comments = $article->comments;
      $user = User::find($article->user_id);
      $country = Country::where('id', $user->country_id)->get()->first();

      return view('articles.show', compact('tags','article',
      'country', 'comments', 'user'));
    }
//articles method
public function articles($id)
{
   $user = User::find($id);
   return view('articles.articles', compact('user'));
}
```

Let us take a close look at the `index()` method first. Think about this line:

```
$articles = Article::all();
```

Eloquent has made querying relationships quite easy. Now through the `Article` model, you can retrieve all the records related to articles. As you learned earlier, Facade provides a `static` interface to classes that are available in the application's service container. Laravel 5.8 ships with many facades that provide access to almost all of Laravel's features. The DB Facade also does the same in all types of database queries. The DB Facade provides methods for each type of query: select, update, insert, and delete.

Tip You can also get all the records by using the `table()` method on DB facade to retrieve the same records. You could have written it directly using the DB facade instead of using the model, as sin `$articles = DB::table('articles')->get();`. Here, you have used the `table` method on the DB facade to begin the query, and the `table` method returns a fluent query builder instance for the given table. The advantage of using DB facade is it allows you to chain more constraints onto the query and then finally get the results using the `get` method.

These three methods are related to three view pages: `articles.articles`, `articles.index`, and `articles.show`. However, before taking a look at the view page code, you will see how in the `Article` model you establish the relationship between different records, as shown here:

```
//code 8.10
//app/Article.php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Article extends Model
{
  protected $fillable = [
      'user_id', 'title', 'body',
  ];

  public function user() {
      return $this->belongsTo('App\User');
  }

  public function users() {
      return $this->belongsToMany('App\User');
  }
  public function tags() {
      return $this->belongsToMany('App\Tag');
  }

  /**
  * Get all of the articles' comments.
  */
  public function comments(){
    return $this->morphMany('App\Comment', 'commentable');
  }
}
```

Each article does not have a complicated relation with the other records as one article has three components attached to it. The first is the user who writes them, the second is the tags the user uses, and the third one is the comments section that has a polymorphic relation with the articles.

Here is the code for the three views of the articles; later, you will add more view pages for inserting or updating records:

```
// resources/views/articles/articles.blade.php
@extends('layouts.app')
@section('content')
<div class="container">
    <div class="row">
        <div class="col-md-8 blog-main col-lg-8 blog-main col-sm-8 blog-main">
          <div class="blog-post">
    <ul class="list-group">
         <div class="panel-heading">All Articles by <a href="/users/{{
         $user->id }}">{{ $user->name }}</a> </div>
                @foreach($user->articles as $article)
                <li class="list-group-item">
                    <h2 class="blog-post-title">
                        <a href="/articles/{{ $article->id }}">{{ $article-
                        >title }}</a>
                    </h2>
                </li>
                @endforeach
    </ul>
            </div>
          <nav class="blog-pagination">
            <a class="btn btn-outline-primary" href="#">Older</a>
            <a class="btn btn-outline-secondary disabled" href="#">Newer
            </a>
          </nav>
        </div>
        <aside class="col-md-4 blog-sidebar">
          <div class="p-3">
              <h3 class="blog-post-title">Know about {{ $article->user-
              >name }}
              </h3>
              <hr class="linenums" color="red">
              <div class="panel panel-default">
                <div class="panel-heading">{{ $article->user->name }}'s
                Profile</div>
                <div class="panel-body">
                    <li class="list-group-item-info">Name : {{ $article-
                    >user->name }}</li>
                    <li class="list-group-item-info">Email: {{ $article-
                    >user->email }}</li>
                    <li class="list-group-item-info">City: {{ $article-
                    >user->profile->city }}</li>
                    <li class="list-group-item-info">About: {{ $article-
                    >user->profile->about }}</li>
                </div>
            </div>
          </div>
        </aside>
    </div>
</div>
@endsection
```

Here is the code for the index page of articles:

```
// resources/views/articles/index.blade.php
@extends('layouts.app')
@section('content')
<div class="container">
    <div class="row">
        <div class="col-md-6 blog-main col-lg-6 blog-main col-sm-6 blog-
        main">
            <div class="blog-post">
    <ul class="list-group">
         @foreach($articles as $article)
         <li class="list-group-item"><h2 class="blog-post-title">
                 <li class="list-group-item"><a href="/articles/{{
                 $article->id }}">{{ $article->title }}</a>
              </h2>
         </li>
         @endforeach
    </ul>
            </div>
          <nav class="blog-pagination">
            <a class="btn btn-outline-primary" href="#">Older</a>
            <a class="btn btn-outline-secondary disabled" href="#">Newer</a>
          </nav>
        </div>
        <aside class="col-md-3 blog-sidebar">
          <div class="p-3">
              <h4 class="font-italic">All Writers</h4>
            @foreach($users as $user)
                 <a href="/users/{{ $user->id }}">{{ $user->name }}</a>...
         @endforeach
          </div>
        </aside>
        <aside class="col-md-3 blog-sidebar">
          <div class="p-3">
              <h4 class="font-italic">Tags-Cloud</h4>
            @foreach($tags as $tag)
                 <a href="/tags/{{ $tag->id }}">{{ $tag->tag }}</a>...
         @endforeach
          </div>
        </aside>
    </div>
</div>
@endsection
```

Finally, here is the show.blade.php code; it is important because it will show a particular article:

```
// resources/views/articles/show.blade.php
@extends('layouts.app')
@section('content')
<div class="container">
    <div class="row">
        <div class="col-md-8 col-md-offset-2">
            <div class="panel panel-default">
                <div class="panel-heading">
                    <h3 class="pb-3 mb-4 font-italic border-bottom">{{
                    $article->title }}</h3>  by
                    <p>{{ $article->user->name }}</p>
                </div>

                <div class="panel-body">
                    <li class="list-group-item">{{ $article->body }}</li>
                    Tags:
                    @foreach($article->tags as $tag)
                    {{ $tag->tag }} ,
                    @endforeach
                    <li class="list-group-item-info">Other Articles by
                        <p>
                            <a href="/users/{{ $article->user_id }}/
                            articles">{{ $article->user->name }}</a>
                        </p>
                        THis user belongs to {{ $country->name }}<p></p>
                    </li>
                    <h3 class="blog-post">
                      All articles from {{ $country->name }}
                    </h3>
                    @foreach($country->articles as $article)
                    <li class="list-group-item">
                      <a href="/articles/{{ $article->id }}">
                        {{ $article->title }}
                      </a>
                    </li>
                    @endforeach
                    <h3 class="blog-post">
                      All comments
                    </h3>
                    @foreach($comments as $comment)
                    <li class="list-group-item">
                      <a href="/comments/{{ $comment->id }}">
                        {{ $comment->body }}
                      </a>
                    </li>
                    @endforeach
                </div>
            </div>
        </div>
    </div>
</div>
@endsection
```

In this `show.blade.php` page, you will find many Eloquent relationship queries where you don't have to add additional constraints; instead, you access the relationship as if it consisted of properties.

For example, in the `ArticleController show($id)` method, you can access them as properties as follows:

```
$tags = Article::find($article->id)->tags;
      $article = Article::find($article->id);
      $comments = $article->comments;
      $user = User::find($article->user_id);
      $country = Country::where('id', $user->country_id)->get()->first();
```

You access all of an article's tags like this:

```
$tags = Article::find($article→id)→tags;
```

This was originally defined in the `Article` and `Tag` models.

Since you have defined the relationship between `Articles` and `Tags` and you have accessed article tags using Eloquent queries, now you can get the related tags in your `show.blade.php` page like this:

```
<li class="list-group-item">{{ $article->body }}</li>
                    Tags:
                    @foreach($article->tags as $tag)
                    {{ $tag->tag }} ,
                    @endforeach
```

Once you get the idea of how the Eloquent queries work and how model relations work, the rest is simple, and any type of complicated tasks can easily be handled.

Likewise, you can now build the `Reviews` part the same way. For brevity, I have not included all the code like with `Articles`. Here I am showing only the code of the `show($id)` method of `ReviewController`, and I show the code for `show.blade.php`:

```
//code 8.11
//app/HTTP/Controllers/ ReviewController.php
    public function show(Review $review)
    {
        $tags = Review::find($review->id)->tags;
        $review = Review::find($review->id);
        $comments = $review->comments;
        $user = User::find($review->user_id);
        $company = Company::find($review->company_id);
        $country = Country::where('id', $user->country_id)->get()->first();
        return view('reviews.show', compact('tags','review',
        'country', 'comments', 'user', 'company'));
    }
```

And you get the values in the `resources/views/reviews/show.blade.php` page, like this:

```
//resources/views/reviews/show.blade.php
@extends('layouts.app')

@section('content')
<div class="container">
    <div class="row">
        <div class="col-md-8 col-md-offset-2">
            <div class="panel panel-default">
                <div class="panel-heading">
                    <h3 class="pb-3 mb-4 font-italic border-bottom">{{
                    $review->title }}</h3>  by
                    <p>{{ $review->user->name }}</p>
                </div>

                <div class="panel-body">
                    <li class="list-group-item">{{ $review->body }}</li>
                    Tags:
                    @foreach($review->tags as $tag)
                    {{ $tag->tag }} ,
                    @endforeach
                    <li class="list-inline">
                      <h3>This review is about
                      the company</h3>
                        <p>
                            <a href="/companies/{{ $review->company_id }}">
                              {{ $company->name }}</a>
                        </p>
                        <p>
                        ***
                        </p>
    <p>
      <cite>
    However, only registered users can view
      the company profile
    </cite>
    </p>
    <p>
      ***
    </p>
</li>
<li class="list-group-item-info">Other Reviews by
    <p>
        <a href="/users/{{ $review->user_id }}/review">
          {{ $review->user->name }}</a>
    </p>
    THis user belongs to {{ $country->name }}<p></p>
</li>
<h3 class="blog-post">
  All reviews from {{ $country->name }}
</h3>
@foreach($country->reviews as $review)
<li class="list-group-item">
  <a href="/reviews/{{ $review->id }}">
    {{ $review->title }}
  </a>
</li>
@endforeach
<h3 class="blog-post">
  All comments
</h3>
@foreach($comments as $comment)
<li class="list-group-item">
  <a href="/comments/{{ $comment->id }}">
    {{ $comment->body }}
  </a>
```

```
                    </li>
                    @endforeach
                </div>
            </div>
        </div>
    </div>
</div>
@endsection
```

These are basically the public sections of your application for anyone to view. I have not covered the inserting and editing parts here. As you progress, you will learn about those parts, but before that, you will see how you can create the companies, projects, and users section, allowing the designated users to insert or edit data.

Home Page, Redirection, and Authentication

Implementing authentication in Laravel is super simple. You have already learned it: you just run the `php artisan make:auth` and `php artisan migrate` commands one after other. These two commands will take care of scaffolding the entire authentication system. Since the authentication process has been configured by default, you need not worry about the registration and login processes that follow it immediately.

Can you tweak the behavior of the authentication service? Yes, you can. The authentication configuration file is `config/auth.php`. However, in most cases, you don't have to customize it. The retrieval of users is done with the help of default providers, and Laravel ships with support for retrieving users using Eloquent and the database query builder.

If this sounds confusing, don't worry. I will again discuss it in a minute.

Let's try to understand the authentication process, step-by-step, first. Laravel comes with two types of authentication drivers: the Eloquent authentication driver and the database authentication driver. If you don't use the Eloquent authentication driver, you need to use the database authentication driver. By default, Laravel includes a `User` model in the `app` directory so that you can get an idea of how to use it with either Eloquent or a database.

If the default database schema works for you, you don't have to change it or add more functionalities in your `users` table. The default database schema looks like this:

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

Now, for your company/project/task management application, I have changed it to this:

```
    Schema::create('users', function (Blueprint $table) {
        $table->increments('id');
        $table->integer('country_id')->nullable();
         $table->integer('role_id')->nullable();
        $table->string('name');
        $table->string('email')->unique();
        $table->timestamp('email_verified_at')->nullable();
        $table->string('password');
        $table->rememberToken();
        $table->timestamps();
    });
}
```

Let's view the code of `HomeController`, as shown here:

```
//code 8.12
//app/HTTP/Controllers/HomeController.php
<?php

namespace App\Http\Controllers;
```

```
use Illuminate\Http\Request;

class HomeController extends Controller
{
    /**
     * Create a new controller instance.
     *
     * @return void
     */
    public function __construct()
    {
        $this->middleware('auth');
    }

    /**
     * Show the application dashboard.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        return view('home');
    }
}
```

The first part of the code is extremely important.

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

This means once you create a `HomeController` instance, the instance invokes a method called `middleware()` and passes an argument called `auth`. Therefore, whatever method follows this constructor method will come under the umbrella of middleware and authentication.

First, the middleware filters the requests, and then authentication starts its workflow, making the application authenticated.

Now, if a guest types the `http://localhost/home` URI in a browser, they will be redirected to the login page. At the same time, all the login-related views are placed in the `resources/views/auth` directory. The `resources/views/layouts` directory is also created at the same time. Although all of these views use the Bootstrap CSS framework, you can tweak them according to your needs.

By default, once a user is authenticated, they are redirected to the `/home` URI. However, you can change this by redefining the `redirectTo` property in the controllers `LoginController`, `RegisterController`, and `ResetPasswordController`.

For brevity, I am showing only the `LoginController` code here. But you need to change the same thing in the two others. Since these three controller classes come under the `Auth` namespace, the `redirectTo` method applies to each one individually.

```
//app/HTTP/Controllers/Auth/LoginController.php
<?php

namespace App\Http\Controllers\Auth;

use App\Http\Controllers\Controller;
use Illuminate\Foundation\Auth\AuthenticatesUsers;

class LoginController extends Controller
{
    /*
    |--------------------------------------------------------------------------
    | Login Controller
    |--------------------------------------------------------------------------
    |
    | This controller handles authenticating users for the application and
    | redirecting them to your home screen. The controller uses a trait
    | to conveniently provide its functionality to your applications.
    |
    */
    use AuthenticatesUsers;

    /**
     * Where to redirect users after login.
     *
     * @var string
     */
    //protected $redirectTo = '/home';
//You can comment out the original one and in the next line change it to
something else
    /**
     * The new redirection of users after login.
     *
     * @var string
     */
      protected $redirectTo = '/';

    /**
     * Create a new controller instance.
     *
     * @return void
     */
    public function __construct()
    {
        $this->middleware('guest')->except('logout');
    }
}
```

Next, you need to modify the handle method in the `RedirectIfAuthenticated` file in `app/HTTP/Middleware/RedirectIfAuthenticated.php` to use your new URI (which is / here) while you redirect the user.

```
//code 8.13
<?php
namespace App\Http\Middleware;
use Closure;
use Illuminate\Support\Facades\Auth;
```

```
class RedirectIfAuthenticated
{
    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @param  string|null  $guard
     * @return mixed
     */
    public function handle($request, Closure $next, $guard = null)
    {
        if (Auth::guard($guard)->check()) {
            //return redirect('/home');
// we have commented out the default redirection and change it to the new one
             return redirect('/');
        }
        return $next($request);
    }
}
```

In this case, the `redirectTo` method will override the `redirectTo` attribute that you changed in the `LoginController.php` code. Before, it was redirected to the /home URI; now it goes to the document root, /. Both redirect to the same URI (here, /).

Before concluding this section, let's see the code of `resources/views/home.blade.php`. Laravel creates it by default.

```
//code 8.14
//resources/views/home.blade.php
@extends('layouts.app')

@section('content')
<div class="container">
    <div class="row justify-content-center">
        <div class="col-md-8">
            <div class="card">
                <div class="card-header">Dashboard</div>
                <div class="card-body">
                    @if (session('status'))
                        <div class="alert alert-success" role="alert">
                            {{ session('status') }}
                        </div>
                    @endif

                    You are logged in!
                </div>
            </div>
        </div>
    </div>
</div>
@endsection
```

The original code is not very long, and it displays a simple message, such as "you are logged in." In this application, I have designed this page in a way so that the user can view the page according to their role.

So, there are many differences between the default `home.blade.page` code, shown next, and the application's home page:

```
//code 8.15
//resources/views/home.blade.php
@extends('layouts.app')

@section('content')
<div class="container">
    <div class="row">
        <div class="col-md-12 col-md-offset-2">
            <div class="panel panel-default">
                <div class="panel-heading">
                <h2 class="blog-post">  Dashboard for Admin </h2>
                </div>
                <div class="panel-body">
                    @if (session('status'))
                        <div class="alert alert-success">
                            {{ session('status') }}
                        </div>
                @endif
                Hello <li class="btn btn-danger">{{ $user->name }}</li>
                You are logged in!
            </div>
            <div class="panel-body">
              <h4 class="blog-title">
                Now you can view, add, edit or delete
                any company, project, and user
              </h4>
            </div>
        </div>
    </div>
</div>
<div class="row">
    <aside class="col-md-4 blog-sidebar">
      <div class="p-3">
          <h3 class="pb-3 mb-4 font-italic border-bottom">
       Add New Companies
      </h3>
        <a href="/companies/create" class="btn btn-primary"
        role="button">Create Companies</a>
        <h4 class="font-italic"><a href="/companies">View All
        Companies</a></h4>
       </div>
    </aside>
     <aside class="col-md-4 blog-sidebar">
      <div class="p-3">
      <h3 class="pb-3 mb-4 font-italic border-bottom">
          Add New Projects
      </h3>
        <a href="/projects/create" class="btn btn-primary"
        role="button">Create Projects</a>
        <h4 class="font-italic"><a href="/projects">View All Projects
        </a></h4>
       </div>
    </aside>
```

```
        <aside class="col-md-4 blog-sidebar">
          <div class="p-3">
              <h3 class="pb-3 mb-4 font-italic border-bottom">
              Add New Users
          </h3>
            <a href="/users/create" class="btn btn-primary"
            role="button">Create Users</a>
            <h4 class="font-italic"><a href="/users">View All Users</a>
            </h4>
          </div>
        </aside>
    </div>
</div>
@endsection
```

You saw this page in Figure 8-1. This home page should allow a registered user to view the dashboard panel. According to the designated role, the user can view, create, edit, or delete data.

There are several ways to handle this task. You will learn about them in the “Role of a User and Authorization” section. In the next section, you will learn about how you can do authorization for a specific role.

Role of a User and Authorization

You have defined a few specific roles, such as administrator, moderator, editor, and member. Each role has specific tasks, such as the administrator can view/create/edit/delete everything including companies, projects, users, reviews, and comments.

You have moderators who can view/create/edit/delete everything except companies. Next, think about the editor. You have decided to let the editors view/create/edit/delete everything, except companies and projects. Finally, consider the task of the general members. They can view/create/edit/delete only reviews and comments.

Keeping every role in mind, now you can attain all these functionalities through `home.blade.php`.

Consider this part of code from the home Blade page:

```
//code 8.16
//resources/views/home.blade.php
<div class="panel-body">
                    @if (session('status'))
                        <div class="alert alert-success">
                            {{ session('status') }}
                        </div>
                    @endif
                    Hello <li class="btn btn-danger">{{ $user->name }}</li>
                    @if(Auth::user()->role_id === 1)
                    <h2 class="blog-post">  Dashboard for Admin </h2>
                    You are logged in as an Administrator!
                    <h4 class="blog-title">
                      Now you can view, add, edit or delete
                      any company, project, and user
                    </h4>
                    <li class="nav-item dropdown">
                        <a id="navbarDropdown" class="nav-link
                        dropdown-toggle" href="#" role="button" data-
                        toggle="dropdown" aria-haspopup="true" aria-
                        expanded="false" v-pre>
                            Companies <span class="caret"></span>
                        </a>
                        <div class="dropdown-menu dropdown-menu-right"
                        aria-labelledby="navbarDropdown">
                            <a href="/companies" class="btn btn-primary"
                            role="button">
                              View All Companies</a>
                            <a href="/companies/create" class="btn btn-
                            primary" role="button">
                              Create Companies</a>
                        </div>
                    </li>
                    <li class="nav-item dropdown">
                        <a id="navbarDropdown" class="nav-link
                        dropdown-toggle" href="#" role="button" data-
                        toggle="dropdown" aria-haspopup="true" aria-
                        expanded="false" v-pre>
                            Projects <span class="caret"></span>
                        </a>
                        <div class="dropdown-menu dropdown-menu-right"
                        aria-labelledby="navbarDropdown">
                            <a href="/projects" class="btn btn-primary"
                            role="button">
                              View All Projects</a>
                            <a href="/projects/create" class="btn btn-
                            primary" role="button">
                              Create Projects</a>
                        </div>
                    </li>
                    <li class="nav-item dropdown">
                        <a id="navbarDropdown" class="nav-link
                        dropdown-toggle" href="#" role="button" data-
                        toggle="dropdown" aria-haspopup="true" aria-
                        expanded="false" v-pre>
                            Users <span class="caret"></span>
                        </a>
                        <div class="dropdown-menu dropdown-menu-right"
                        aria-labelledby="navbarDropdown">
                            <a href="/users" class="btn btn-primary"
                            role="button">
                              View All Users</a>
                        </div>
//code is incomplete
```

It assures that the administrator can now do every operation, and once the user logs in, according to the assigned role of an administrator, the page looks like Figure 8-1.

Note You can add reviews and comments here on your own. Create the models first and then define the model relations. Next, create controllers and views accordingly, as you've learned here.

If the user is a moderator, as I have pointed out before, the moderator can do every operation except the "companies" part. When the moderator signs in, the look of the dashboard looks like Figure 8-2.

I have limited the functionalities of the moderator or editor to projects and users; by following the same rule, you can add reviews and comments for them.

In the `home.blade.php` code, this logic is important as it defines the main logic of separations:

```
@if(Auth::user()->role_id === 1);
```

It is clear that if the user doesn't have the role ID 1 (that is, if the user is not an administrator), they cannot access this part. And this section of logic has been followed by this conditional:

```
@elseif(Auth::user()->role_id === 2)
```

This states that the user must have role ID 2; that is, they need to be a moderator.

Continuing this logic, you can continue developing your application and define and separate the activities of editors and general members by adding some extra functionality here and there.

Authorization Through the Blade Template

When you build a web application, forms and HTML play important roles.

The administrator should be able to create companies and edit any company. The moderator should have the same ability to create or edit any project.

To create this functionality, you need to have necessary forms and HTML elements in the view Blade pages. Figure 8-8 shows what it looks like when the administrator logs in and tries to insert data.

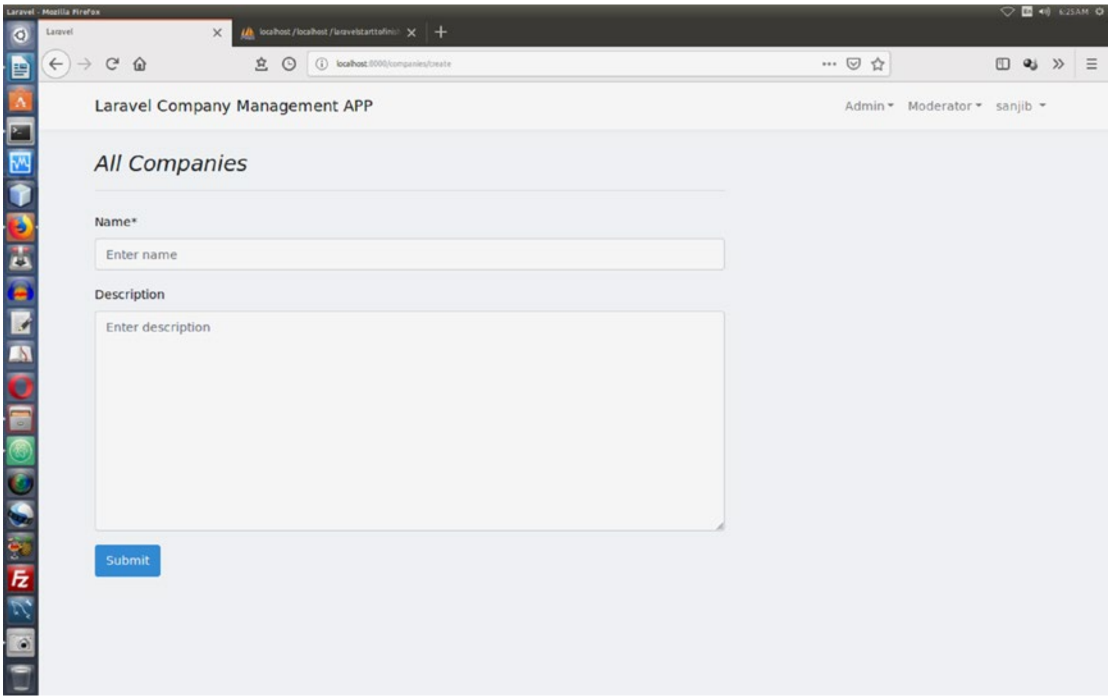


Figure 8-8. *The dashboard for administrator to insert data in companies page*

First, let's see how you can add functionality to the company controller so that you can insert, edit, or delete data into the companies' database with the help of the company model and view pages.

Before creating a companies page to show, edit, and delete data, you need to fill in the companies table with some data. In the final application, the administrators would do this. Currently, you can either use your terminal or, if you want, use the `phpMyAdmin` interface. Or you can use Tinker to view or manipulate your data. In this application, I have already used `Faker` and have inserted data for about 20 fake companies. As you saw earlier, you can use `Faker` to add any kind of data, be it articles, users, or anything that you need to test your application; now you can add some company data the same way.

You have to create a folder called `companies` inside the `resources/views` first. The question is, what types of Blade pages are required? You can guess it from the company controller. Since I have discussed how HTTP verbs, URIs, action methods, and route names are linked together, you can guess which action methods you should use to reach your destination view pages. The following company controller is a blank page. You need to add functionalities here so that you can continue.

There are seven methods that your make:controller -resource command has created. They are self-explanatory. Through the index() method, you can show the front page of any company. You can use the show() method for any other purposes. There are create() and store() methods for inserting new companies' data. The edit() method will take you to the update() method where the administrators can update any company data. Finally, there is the delete() method to remove any data permanently.

So, inside the companies view page, you will create four pages now: index.blade.php, show.blade.php, edit.blade.php, and create.blade.php. To start with, let's concentrate on index.blade.php . This page will show every company name to everyone. You don't want any administrator actions here. However, you want only registered visitors to be able to click each company name and see the details on another page. You will use the show.blade.php page for that purpose; however, you don't want any guest viewer to be able to view those pages. You also want the administrator to be able to handle all operations regarding all the companies' pages. There are four roles you have set so far: administrator, moderator, editor, and general members or users who are assigned tasks either by the administrator or moderator. The editor's role should be restricted to only editing; in other words, an editor can edit a user's blogs and comments.

In the next section, you will take a look at all the view pages. Before that, you need to understand how you should make resourceful controllers restrict the movement inside the company pages. Here is the code for that:

```
// code 8.17
// app/HTTP/Controllers/CompanyController.php
<?php

namespace App\Http\Controllers;
use App\User;
use App\Company;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;

class CompanyController extends Controller
{
    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
```

```
public function index()
{
    if( Auth::check() ){
        $companies = Company::where('user_id', Auth::user()->id)->get();

        if(Auth::user()->role_id == 1){
            return view('companies.index', ['companies'=> $companies]);
        }
    }
    return view('auth.login');
}

/**
 * Show the form for creating a new resource.
 *
 * @return \Illuminate\Http\Response
 */
public function create()
{
  if( Auth::check() ){
    if(Auth::user()->role_id == 1){
            return view('companies.create');
    }
  }
    return view('auth.login');

}

/**
 * Store a newly created resource in storage.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return \Illuminate\Http\Response
 */
```

```
    public function store(Request $request)
    {
        if(Auth::check()){
            $company = Company::create([
                'name' => $request->input('name'),
                'description' => $request->input('description'),
                'user_id' => Auth::user()->id
            ]);
            if($company){
                return redirect()->route('companies.show', ['company'=>
                $company->id])
                        ->with('success' , 'Company created successfully');
            }

        }

            return back()->withInput()->with('errors', 'Error creating new
            company');
    }
    /**
     * Display the specified resource.
     *
     * @param  \App\Company  $company
     * @return \Illuminate\Http\Response
     */
    public function show(Company $company)
    {
      if( Auth::check() ){
        if(Auth::user()->role_id == 1){
            $company = Company::find($company->id);
            return view('companies.show', ['company' => $company]);
        }
      }
        return view('auth.login');
    }
```

```
/**
 * Show the form for editing the specified resource.
 *
 * @param  \App\Company  $company
 * @return \Illuminate\Http\Response
 */
public function edit(Company $company)
{
  if( Auth::check() ){
    if(Auth::user()->role_id == 1){
    $company = Company::find($company->id);
    return view('companies.edit', ['company' => $company]);
  }
}
}
/**
 * Update the specified resource in storage.
 *
 * @param  \Illuminate\Http\Request  $request
 * @param  \App\Company  $company
 * @return \Illuminate\Http\Response
 */
public function update(Request $request, Company $company)
{
    $updateCompany = Company::where('id', $company->id)->update(
            [
                'name'=> $request->input('name'),
                'description'=> $request->input('description')
            ]
    );
  if($updateCompany){
      return redirect()->route('companies.show', ['company'=> $company-
      >id])
      ->with('success' , 'Company updated successfully');
  }
```

```
      //redirect
      return back()->withInput();
    }

    /**
     * Remove the specified resource from storage.
     *
     * @param  \App\Company  $company
     * @return \Illuminate\Http\Response
     */
    public function destroy(Company $company)
    {
    }
    public function getDestroy($id)
    {
        $company = Company::findOrFail($id);
        if($company->destroy($id)){
            return redirect()->route('companies.index')->with('success' ,
            'Company deleted successfully');
        }
    }
}
```

Let's take a look at the index method. It takes users to the `companies.index` page.

```
        public function index()
    {
        if( Auth::check() ){
            $companies = Company::where('user_id', Auth::user()->id)->get();

            if(Auth::user()->role_id == 1){
                return view('companies.index', ['companies'=> $companies]);
            }
        }
        return view('auth.login');
    }
```

The Auth class uses two static methods, check() and user(). I have used the necessary namespace so that it can do that. At the top of this file, you will find these lines of code:

```
namespace App\Http\Controllers;
use App\User;
use App\Company;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;
```

You need two models primarily: User and Company. You will learn about the Auth facade in the next chapter. However, primarily, you need to keep one thing in mind. The first condition uses Auth::check();. If the viewer is not a registered user, this checking process restricts the user's movement. In the second condition, you check whether the user has a role_id value of 1, which belongs only to an administrator. Regarding every operation concerning companies, you always keep checking that these two conditions are true. If not, the application will take the visitor to the login page.

Let's see the companies.create Blade code now:

```
//code 8.18
//resources/views/companies/create.blade.php
@extends('layouts.app')

@section('content')


<div class="container">
    <div class="row">
        <div class="col-md-8 blog-main col-lg-8 blog-main col-sm-8 blog-
        main">
          <h3 class="pb-3 mb-4 font-italic border-bottom">
            All Companies
          </h3>
            <div class="blog-post">
            <h2 class="blog-post-title"></h2>
             <form method="post" action="{{ route('companies.store') }}">
             {{ csrf_field() }}
            <div class="form-group">
            <label for="company-name">Name<span class="required">*</span>
            </label>
            <input   placeholder="Enter name"
            id="company-name"
            required
            name="name"
            spellcheck="false"
            class="form-control"
            />
            </div>
            <div class="form-group">
            <label for="company-content">Description</label>
            <textarea placeholder="Enter description"
            style="resize: vertical"
            id="company-content"
            name="description"
            rows="10" spellcheck="false"
            class="form-control autosize-target text-left">
            </textarea>
            </div>
            <div class="form-group">
            <input type="submit" class="btn btn-primary"
            value="Submit"/>
            </div>
            </form>
            </div>
        </div>
    </div>
</div>
@endsection
```

Opening and closing a form in any Laravel's application is quite easy.

```
{{ Form::open(array('url' => 'companies/create')) }}
    //
{{ Form::close() }}
```

A POST method will be assumed by default; however, you are always free to add other functionalities like PUT or PATCH.

```
echo Form::open(array('url' => 'companies/create', 'method' => 'put'))
```

Here I have followed the conventional method:

```
<form method="post" action="{{ route('companies.store') }}">
             {{ csrf_field() }}
```

You are always free to choose your own method. However, in the previous line, the `route()` method is important because it points to the `store()` method of `CompanyController.php` file, as shown here:

```
    //app/HTTP/Controllers/CompanyController.php
public function store(Request $request)
    {
        if(Auth::check()){
            $company = Company::create([
                'name' => $request->input('name'),
                'description' => $request->input('description'),
                'user_id' => Auth::user()->id
            ]);
            if($company){
                return redirect()->route('companies.show', ['company'=>
                $company->id])
                        ->with('success' , 'Company created successfully');
            }
        }
            return back()->withInput()->with('errors', 'Error creating new
            company');
    }
```

This completes the whole insertion process. The editing part is almost the same, except a few changes. Let's see what the `edit.blade.php` page code looks like:

```
// resources/views/companies/edit.blade.php
@extends('layouts.app')
@section('content')
```

```
<div class="container">
    <div class="row">
        <div class="col-md-8 blog-main col-lg-8 blog-main col-sm-8
        blog-main">
          <h3 class="pb-3 mb-4 font-italic border-bottom">
            Edit Companies
          </h3>
            <div class="blog-post">
    <ul class="list-group">
         <form method="post" action="{{ route('companies.update',
         [$company->id]) }}">
                            {{ csrf_field() }}

                            <input type="hidden" name="_method" value="put">

                            <div class="form-group">
                                <label for="company-name">Name<span
                                class="required">*</span></label>
                                <input   placeholder="Enter name"
                                          id="company-name"
                                          required
                                          name="name"
                                          spellcheck="false"
                                          class="form-control"
                                          value="{{ $company->name }}"
                                           />
                            </div>
                            <div class="form-group">
                                <label for="company-content">Description
                                </label>
                                <textarea placeholder="Enter description"
                                          style="resize: vertical"
                                          id="company-content"
                                          name="description"
                                          rows="10" spellcheck="false"
                                          class="form-control autosize-
                                          target text-left">
```

```
                                                {{ $company->description }}
                                                </textarea>
                                </div>
                                <div class="form-group">
                                    <input type="submit" class="btn btn-primary"
                                           value="Submit"/>
                                </div>
</form>
    </ul>
            </div>

        </div>
    </div>
</div>
@endsection
```

Here the following line is extremely important:

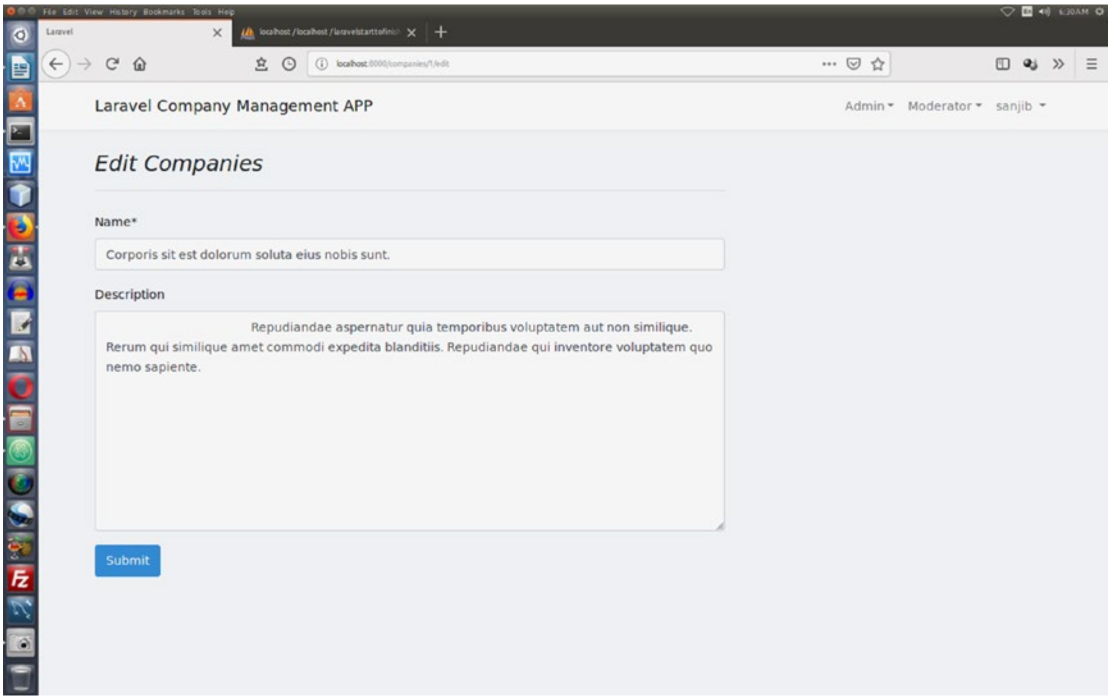
```
<form method="post" action="{{ route('companies.update',[$company->id])
}}"> {{ csrf_field() }}
```

The route() method passes the update() method of CompanyController.php, and that code looks like this:

```
    //app/HTTP/Controllers/CompanyController.php
public function update(Request $request, Company $company)
    {
        $updateCompany = Company::where('id', $company->id)->update(
                [
                    'name'=> $request->input('name'),
                    'description'=> $request->input('description')
                ]
        );
      if($updateCompany){
          return redirect()->route('companies.show', ['company'=>
          $company->id])
          ->with('success' , 'Company updated successfully');
      }
```

```
        //redirect
        return back()->withInput();
    }
```

The edit companies page looks like Figure 8-9. On this edit page, only the administrator has the power to enter and edit anything.


Laravel Company Management APP

Edit Companies

Name*

Corporis sit est dolorum soluta eius nobis sunt.

Description

Repudiandae aspernatur quia temporibus voluptatem aut non similique. Rerum qui similique amet commodi expedita blanditiis. Repudiandae qui inventore voluptatem quo nemo sapiente.

Figure 8-9. *Edit companies page*

Based on the same techniques, you can edit and update any data using controller, model, and view pages.

If you want to insert or edit data, you need to make yourself authorized to do that. You have seen how Laravel has helped you achieve your goal in a simple way, without writing hundreds of lines of code.

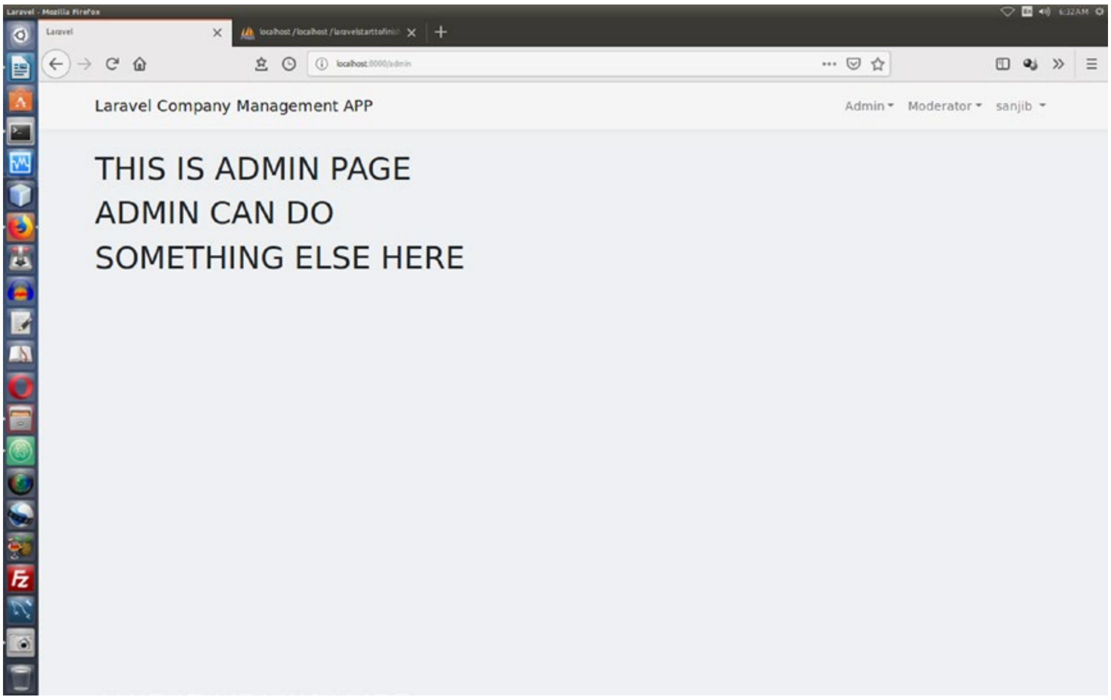
Laravel has another authorization technique that you can also utilize. In the next section, you will learn about it.

Implementing Authorization Using Gates and Policies

You have seen how authentication services come out of the box, and in this section I will discuss authentication in more detail. Laravel provides a simple way to authorize user actions against a given resource. To authorize actions, you use gates and policies. Let's first see them in action, and after that you will learn how to use them. In Figure 8-1, at the beginning of the chapter, you saw what the home page looks like after the administrator has logged in.

In the top-right corner, you can see another Admin link. It does not show up when other users log in. I have used a simple authorization technique to make it possible. In the cases of moderators, editors, or general members, you can also have similar things in place. For example, when a moderator logs in, you can display a Moderator link to take the moderator to a destination that is reserved for their own consumption. Other users won't see that link.

Let's see what happens when the administrator clicks the Admin link; see Figure 8-10.



THIS IS ADMIN PAGE
ADMIN CAN DO
SOMETHING ELSE HERE

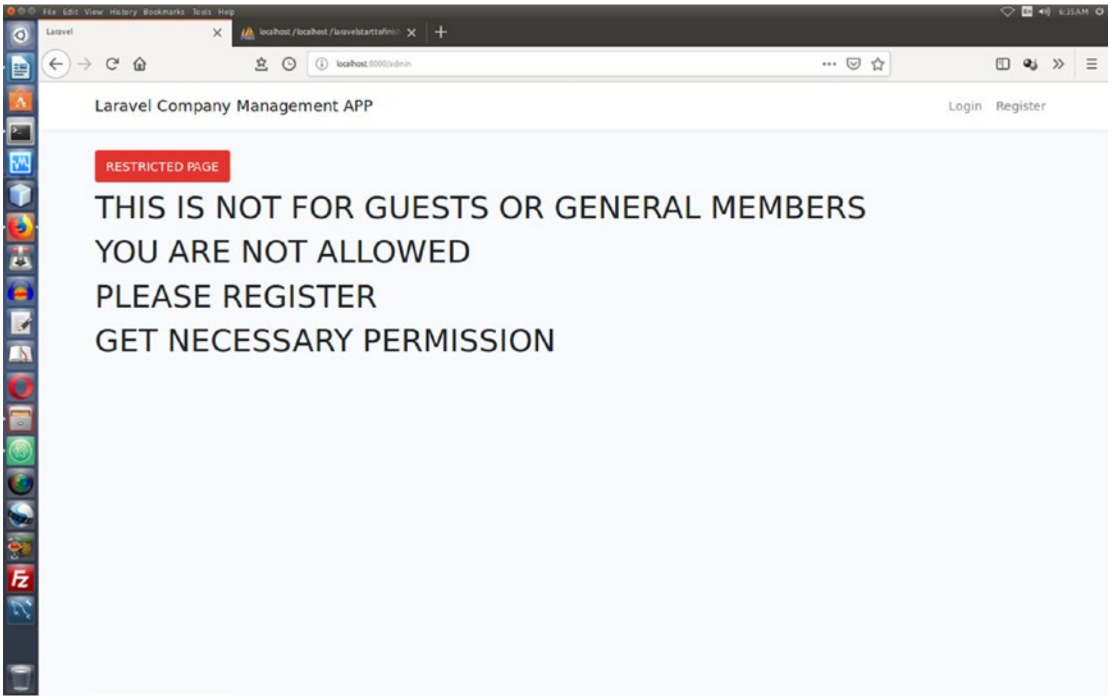
Figure 8-10. *The dashboard for administrator after authorization takes place*

As you can see, you can use this administrator page for different tasks than just inserting and updating companies, projects, tasks, and user data. The next pages will show some other types of administrator pages that I created with gates and policies.

In Figure 8-10, the URI is `http://localhost:8000/admin`. Whenever the administrator signs in, they are taken to this particular destination.

What happens when someone who is not an administrator tries to type the same URI in the browser and wants to penetrate the site?

Well, Figure 8-11 shows the result.



THIS IS NOT FOR GUESTS OR GENERAL MEMBERS
YOU ARE NOT ALLOWED
PLEASE REGISTER
GET NECESSARY PERMISSION

Figure 8-11. *When someone other than administrator wants to view admin page*

This means the URI has been filtered automatically, and a restriction has taken place. Therefore, by applying simple authorization techniques, you have achieved many things in one go. You have created a system where only the administrator can view the Admin link and where only administrators can reach the administrator page meant for them. Moreover, the administrator page has been filtered automatically.

Let's see how to do this.

How Authorization Works

First, you need to add one column in your `users` table. I have added a column called `admin` (tinyint) and made its default value 0 so that I can change it to 1 for the administrator.

The SQL query that you can run is this:

```
ALTER TABLE `users` ADD `admin` TINYINT NOT NULL DEFAULT '0' AFTER
`updated_at`;
```

Now you can think of gates and policies as your routes and controllers. While gates gives you a closure-based approach toward authorization, the policies provide a controller like logic handling. However, I would like to add one important statement here. It is not mandatory that you have to follow this approach for building an authorization mechanism. You could have taken the role-based approach as well even using the Blade template.

So, as mentioned, your first step is to add a column in your `users` table. You can name it anything (I've named it `admin`), but whatever name you assign, you need to use that name in your gates and policies.

Next, open the `app/Providers/AuthServiceProvider.php` file and add this line:

```
//code 8.19
//app/Providers/AuthServiceProvider.php
    /**
     * Register any authentication / authorization services.
     *
     * @return void
     */
    public function boot()
    {
        $this->registerPolicies();

        Gate::define('admin-only', function ($user) {
          if($user->admin == 1){
            return TRUE;
          }
          return FALSE;

        });
    }
```

As I said earlier, gates work as routes; hence, the closure defines that if the `user` object that accesses the `admin` property (remember, you have added the column `admin` in `users` table, so the name matters) equals 1, it will return `TRUE`. Otherwise, it returns `FALSE`.

Next, you need to fix that gates in the `route` file, which is `routes/web.php`.

```
//code 8.20
//routes/web.php
Route::get('/admin', function () {
  if (Gate::allows('admin-only', Auth::user())) {
        // The current user can view this page
        return view('admin');
    }
    else{
      return view('restrict');
    }
});
```

Here you should keep one thing in mind. I have added the `admin` column in the `users` table, so the gates will allow admins only. If you had named the column something else, such as `isadmin`, then your gates would have allowed "isadmin" only. That is the rule.

Now, the user who has their admin column set to 1 can view the `http://localhost/admin` URI. Anybody else will land at the `http://localhost/restrict` page. Through the gates closures, you can easily determine whether a user is authorized to perform a given action. Here, in this company/project/task management application, since the user `sanjib` is the administrator, his role has been defined in the `App\Providers\AuthServiceProvider` class using the `Gates` facade. As gates always receive a user instance as their first argument, it is easy to determine whether `$user->admin == 1`.

Once the gates have been defined, you can use the `allows` or `denies` method. Laravel will automatically take care of passing the user into the gate closures. If you want to update a post, you can write it like this:

```
    if (Gate::allows('update-post', $post)) {
        // The current user can update the post...
    }
/* or you can deny any user from doing it */
    if (Gate::denies('update-post', $post)) {
        // The current user can't update the post...
    }
```

Now you're left with another task. How you can determine the role of the user in your Blade template? How you can use this `Gates` facade so that the administrator alone can view the `admin` link?

Since the right side of the "navigation bar" has been defined in the `resource/views/layouts/app.blade.php` file, you can use a three-line code inside this navigation bar.

```
//code 8.21
// resource/views/layouts/app.blade.php
<!-- Right Side Of Navbar -->
                    <ul class="navbar-nav ml-auto">
                        <!-- Authentication Links -->
                        @guest
                            <li class="nav-item">
                                <a class="nav-link" href="{{ route('login')
                                }}">{{ __('Login') }}</a>
                            </li>
                            <li class="nav-item">
                                @if (Route::has('register'))
                                    <a class="nav-link" href="{{
                                    route('register') }}">{{ __('Register')
                                    }}</a>
                                @endif
                            </li>
```

```
    @else
    <li class="nav-item dropdown">
      @can('admin-only', Auth::user())
            <a id="navbarDropdown" class="nav-link
            dropdown-toggle" href="/admin">Admin</a>
      @endcan
    </li>
    <li class="nav-item dropdown">
            <a id="navbarDropdown" class="nav-link
            dropdown-toggle"
            href="#" role="button" data-
            toggle="dropdown" aria-haspopup="true"
            aria-expanded="false" v-pre>
                {{ Auth::user()->name }} <span
                class="caret"></span>
            </a>

            <div class="dropdown-menu dropdown-menu-
            right" aria-labelledby="navbarDropdown">
                <a class="dropdown-item" href="{{
                route('logout') }}"
                   onclick="event.preventDefault();
                                  document.getElement
                                  ById('logout-form').
                                  submit();">
                    {{ __('Logout') }}
                </a>

                <form id="logout-form" action="{{
                route('logout') }}" method="POST"
                style="display: none;">
                    @csrf
                </form>
            </div>
        </li>
    @endguest
</ul>
```

This code determines what a guest would view in the upper-right navigation and what the registered administrator could view. This simple logic has been handled by this line:

```
<li class="nav-item dropdown">
                          @can('admin-only', Auth::user())
                                <a id="navbarDropdown" class="nav-link
                                dropdown-toggle" href="/admin">Admin</a>
                          @endcan
                        </li>
```

In the Blade templates, you are displaying a portion of the page only if the user is authorized to perform a given action. If you want the users to update the post, you can use the `@can` and `@cannot` family of directives, as you can see in the previous code.

How Policies Work

You have so far been able to make the administrator view the restricted pages designated for administrators only. Now you will learn to enhance this capacity. An administrator will view the designated page for the moderator as well. Not only that, you will see how to add the same facility for the moderator so that the moderator can view the link of the restricted page for the moderators.

You can apply the same technique as you have adopted for the administrator. So, in the `//app/Providers/AuthServiceProvider.php` file, you can add the same functionalities, as shown here:

```
 /**
     * Register any authentication / authorization services.
     *
     * @return void
     */
    public function boot()
    {
        $this->registerPolicies();

        Gate::define('admin-only', function ($user) {
          if($user->admin == 1){
            return TRUE;
          }
          return FALSE;

        });

         Gate::define('mod-only', function ($user) {
          if($user->mod == 1){
            return TRUE;
          }
          return FALSE;

        });
    }
```

In your routes/web.php file, you should define the closure this way:

```
Route::get('/mod', function () {
  if (Gate::allows('mod-only', Auth::user())) {
        // The current user can view this page
        return view('mod');
    }
    else{
      return view('restrict');
    }
});
```

As expected, now this mechanism works fine for your application. The administrator (here sanjib) now can view the Admin link and the Moderator link in the top-right navigation bar.

This action was caused by a little change of code in the source layout page, resources/views/layouts/app.blade.php, as shown here:

```
<li class="nav-item dropdown">
                          @can('admin-only', Auth::user())
                                <a id="navbarDropdown" class="nav-link
                                dropdown-toggle" href="/admin">Admin</a>
                          @endcan
                        </li>
```

```
<li class="nav-item dropdown">
  @can('mod-only', Auth::user())
        <a id="navbarDropdown" class="nav-link
        dropdown-toggle" href="/mod">Moderator</a>
  @endcan
</li>
```

I have also added a new column `mod` in the `users` table and turned it on by changing the default value from 0 to 1.

Now if a moderator signs in, they can also view the moderator link in the top-right navigation bar (Figure 8-2).

The specialty of this page is that the moderator cannot view the Admin link, of course. At the same time, the administrator can view everything and access everything.

Now as far as your application logic works, everything goes perfectly. However, you could have made the same functionalities much tidier using policies. So far you have added more functionalities in our `//app/Providers/AuthServiceProvider.php` file and it works. However, this is not wise to have crowded all the application logic in your `//app/Providers/AuthServiceProvider.php` file, making it unnecessary long and clumsy.

Laravel comes with policies that might define the gates in advance so that you can just register your policies in your gates and get the same result.

Why Are Policies Needed?

To answer this question, you need some classes where you can organize your authorization logic around a particular model or resource. You want to create separate authorization logic for the administrator and the moderator. Both belong to the `user` model. In your route, you are not going to use any resource for them because you want to keep them separate through closure.

So for the administrator, you have admin policies, and for the moderator you need `mod` policies. Creating the policies is simple.

```
$ php artisan make:policy admin
Policy created successfully.
$ php artisan make:policy mod
Policy created successfully.
```

Now you have two files created automatically, `App/Policies/admin.php` and `App/Policies/mod.php`, by the `artisan` commands. You could have associated the policies with a model also.

```
php artisan make:policy admin --model=User
```

In both cases, the generated policies will be placed in the `app/Policies` directory. If this directory does not exist in your application, Laravel will create it.

The generated policy class would be an empty class if you hadn't used the model associated with it. If you had specified the particular `--model`, the basic CRUD policy methods would already be included in the class.

Now in the `App/Policies/admin.php` class, you are going to organize the application logic for the user who is the administrator. At the same time in the `App/Policies/mod.php`, you will organize the application logic for the user who is the moderator.

Let's look at the `App/Policies/admin.php` code here:

```
//code 8.22
//App/Policies/admin.php
<?php

namespace App\Policies;

use App\User;
use Illuminate\Auth\Access\HandlesAuthorization;

class admin
{
    use HandlesAuthorization;

    /**
     * Create a new policy instance.
     *
     * @return void
     */
    public function __construct()
    {
        //
    }
    public function admin_only($user)
    {
        if($user->admin == 1){
            return TRUE;
          }
          return FALSE;
    }
}
```

You have taken the authorization logic from your gates and kept it inside the `admin_only` method. You have also passed the `user` object so that you can use the `User` model and its table attributes.

The same thing happens in the case of the `mod` policies, as shown here:

```
//code 8.23
//App/Policies/mod.php
<?php

namespace App\Policies;

use App\User;
use Illuminate\Auth\Access\HandlesAuthorization;

class mod
{
    use HandlesAuthorization;

    /**
     * Create a new policy instance.
     *
     * @return void
     */
    public function __construct()
    {
        //
    }
    public function mod_only($user)
    {
        if($user->mod == 1){
            return TRUE;
          }
          return FALSE;
    }
}
```

Now the time has come to register the policies. The `app/Policies/AuthServiceProvider` is included with the fresh Laravel installation. This `AuthServiceProvider` maps the Eloquent models to the corresponding policies. Once you have registered the policies in `AuthServiceProvider`, the application will start instructing Laravel which policy to utilize while authorizing actions against a given model.

In `AuthServiceProvider`, you will map the policies in this way:

```
protected $policies = [
        'App\Model' => 'App\Policies\ModelPolicy',
        'App\User' => admin::class,
        'App\User' => mod::class
    ];
```

After that, you will register the services.

```
    public function boot()
    {
        $this->registerPolicies();

        Gate::define('admin-only', 'App\Policies\admin@admin_only');


        Gate::define('mod-only', 'App\Policies\mod@mod_only');
    }
```

The full `app/Policies/AuthServiceProvider.php` code looks like this:

```
//code 8.24
//app/Policies/ AuthServiceProvider.php
<?php
```

```
namespace App\Providers;

use App\Policies\admin;
use App\Policies\mod;
use App\User;
use Illuminate\Support\Facades\Gate;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as
ServiceProvider;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * The policy mappings for the application.
     *
     * @var array
     */
    protected $policies = [
        'App\Model' => 'App\Policies\ModelPolicy',
        'App\User' => admin::class,
        'App\User' => mod::class
    ];

    /**
     * Register any authentication / authorization services.
     *
     * @return void
     */
    public function boot()
    {
        $this->registerPolicies();

        Gate::define('admin-only', 'App\Policies\admin@admin_only');

        Gate::define('mod-only', 'App\Policies\mod@mod_only');
    }
}
```

Once the policies have been mapped, you have registered the methods that will be responsible for each action this 'policy' authorizes.

Tips Laravel provides authentication services out of the box. In fact, you can manage the whole authorization process through the roles assigned to the corresponding users. However, it is always wise to utilize the advantages of Laravel's default authorization services also by using gates and policies. Authorizing given actions against a resource is simple and does not take much time.

Before using gates and policies, you need to alter your `users` table with a SQL query like the following because you have used a different `users` table so far. However, now you need two new columns, `admin` and `mod` for the administrator and moderators, respectively.

```
ALTER TABLE `users` ADD `admin` TINYINT NOT NULL DEFAULT '0'
AFTER `updated_at`;

ALTER TABLE `users` ADD `mod` TINYINT NOT NULL DEFAULT '0'
AFTER `admin`;
```

This means you can set the default 0 value to 1 for the desired candidates.

CHAPTER 9

Containers and Facades

Ever since Laravel 4, you have been encouraged to follow the SOLID design principle when creating applications, and this is even more true in Laravel 5.8 so that you can avoid hard-coding and can write cleaner code. Let's see what the SOLID design principle is all about.

This chapter does not have enough pages to give a detailed description of the SOLID principle, but I will discuss it in a nutshell in the first section.

Creating any application in Laravel always needs lots of class dependencies. You may use third-party packages, such as Carbon to manipulate the date and time, or you may have your own class repositories to be injected into the application; whatever the reasons are, you will want a tool to manage these operations seamlessly. The Laravel service container is a powerful tool that manages such class dependencies and performs dependency injection, and the SOLID design principle is a kind of theoretical axiom on which the Laravel service container and facade concepts rely heavily. Facades are connected to the service containers because facades provide static interfaces to the classes that are available in the service containers.

SOLID Design Principle

SOLID actually consists of five design principles as articulated by Robert "Uncle Bob" Martin.

SOLID stands for

- Single responsibility principle
- Open-closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle

© Sanjib Sinha 2019
S. Sinha, *Beginning Laravel*, https://doi.org/10.1007/978-1-4842-4991-8_9

Single Responsibility Principle

The single responsibility principle means a class should have one, and only one, reason to change.

In other words, in a Laravel application, a controller class handles only one resource, which is connected to one corresponding model. As you may have noticed in the applications in this book, you have maintained this single responsibility principle without even trying.

Limiting class knowledge is important. The class's scope should be narrowly focused. A class would do its job and not at all be affected by any changes that take place in its dependencies.

Remember, if you can build a library of small classes with well-defined responsibilities, your code will be more decoupled and easier to test and run.

The Open-Closed Principle

The open-closed principle means a class is always open for extension but closed for modification.

Why is that?

Only you can make some changes in behavior. Unless it is a behavior change, you should not modify your source code. If you can do your job without touching the source code, then you are following the open-closed principle. You should separate extensible behavior behind an interface and flip the dependencies. In other words, while creating an application, you should maintain the data abstraction principle, which is the main principle of any object-oriented programming language. Using an interface between two classes always separates those classes in such a way that one class does not know the other class's intention.

Any time you modify your code, there is a possibility of breaking the old functionality or even adding new bugs. But if you can plan your application in the beginning based on the open-closed principle, you can modify your code base as quickly as possible without affecting it negatively. Again, if you don't use an interface between two classes, the extensibility of an application can be jeopardized.

Liskov Substitution Principle

This principle may sound intimidating, but it is extremely helpful and easy to understand. It says that derived classes must be substitutable for their base class, which means objects should be replaceable with instances of their subtypes without altering the correctness of the program.

As mentioned, the principle says that objects of a superclass will be replaceable with objects of its subclasses without breaking the application. That requires the objects of your subclasses to behave in the same way as the objects of your superclass. This is a guiding principle of any object-oriented programming and fundamental paradigm that results in proper inheritance.

The Interface Segregation Principle

The interface segregation principle is all about singular responsibilities. In a nutshell, it says that an interface should be granular and focused. No implementation of an interface should be forced to implement methods that it does not use. Accordingly, break functionality into small interfaces where needed for your implementation. Plan this out before creating an application and enjoy the decoupled easygoing ride of the interface segregation principle.

Dependency Inversion Principle

Finally, the dependency inversion principle states that high-level code should not depend on low-level code. Instead, the high-level code should depend on abstraction that acts as a middleman between the high level and the low level. The second aspect is that abstraction does not depend upon details; instead, the details depend upon abstractions.

In a Laravel application, you may have noticed that functionalities are given to the users, but you always hide the implementation details from the users. For example, in a view page when you call any object properties and methods, the user does not know where the data comes from. The business logic is hidden in the model that deals with the abstraction. The controller class is in between the logic and the model and controls the high-level code. The low-level code in the view pages depends on the controller, but the inverse is not true here.

Interfaces and Method Injection

Abstraction in OOP involves the extraction of relevant details. Consider the role of a car salesperson. There are many types of consumers. Everyone wants to buy a car, no doubt, but each one has differences in their criteria. Each of them is interested in one or two certain features. This attribute varies accordingly. Shape, color, engine power, power steering, price…the list is endless. The salesperson knows all the details of the car but does he repeat the list one by one until someone lands on their choice?

No.

He presents only the relevant information to each potential customer. As a result, the salesperson practices abstraction and presents only the relevant details to each customer. Now consider abstraction from the perspective of a programmer who wants a user to add items to a list.

Abstraction does not mean that information is unavailable, but it assures that the relevant information is provided to the user. For example, when a user adds items to a shopping cart, the user handles the functionalities only and is completely unaware of the implementations. So, only the relevant information is present or displayed in any application for the users who use those details to complete the tasks.

PHP 5 introduced abstract classes and methods, and PHP 7 enhances them, making the general-purpose language completely object-oriented. Classes defined as abstract cannot be instantiated, and any class that contains at least one abstract method must also be abstract.

Remember that abstract methods cannot define the implementation. On the other hand, object interfaces allow you to create code that specifies which methods a class must implement, without having to define how these methods are handled. So, you can define an abstract cart that specifies what all carts can do, not how they do it. You then let the different e-commerce platforms create their own custom carts that specify how to carry out a cart's functionality. These custom carts are interchangeable because they all adhere to the interface defined by the abstract cart.

Interfaces are defined with the `interface` keyword, in the same way as a standard class, but without any of the methods having their contents defined.

All methods declared in an interface must be public; this is the nature of an interface. In Laravel, you will find the injection of interfaces to the classes frequently. In Laravel, an interface is considered to be a *contract*. In your application you can also do that if needed; I will show this technique in detail later in this chapter.

But contract between whom? And why? An interface does not contain any code; it only defines a set of methods that an object implements. With regard to the SOLID design principle, I have talked about maintaining a library of small classes with clearly defined scopes, which is achievable with the help of interfaces.

Contracts vs. Facades

You know that the Laravel 5.8 framework depends on many blocks of interfaces, classes, and other packaged versions that package developers have developed. Laravel 5.8 has happily used them, and I encourage you to do the same by following the SOLID design principle and using loose coupling. To master the framework properly, you need to understand the core services that run Laravel 5.8.

What are the main resources behind the scenes? Basically, contracts come in between the classes, regarding this scenario. Contracts are interfaces that provide this service to make an application loosely coupled. For example, `Illuminate\Contracts\Mail\Mailer` defines the process of sending e-mails, so this interface simply pools the implementation of mailer classes powered by SwiftMailer.

Now what are facades? Do they have any similarity or relationship to anything else? First, you should know that facades are also interfaces. But they have a distinct difference from contracts.

Facades are static interfaces that supply methods to the classes of a service container. You have seen a lot of facades already, such as `App`, `Route`, `DB`, `View`, and so on. The main aspect of facades is that you can use them without type-hinting. Like in your `routes.php` file, you can write this:

```
//code 9.1
Route::bind('books', function ($id){
return App\Book::where('id', $id)-->first();
});
```

This `Route` facade directly uses the contracts out of the service container. Though facades are static interfaces, they have more expressive syntax and provide more testability and flexibility than a traditional static methodology. But the advantage of `Contracts` is that you can define explicit dependencies for your classes and make your application more loosely coupled.

Of course, for most applications, facades work fine. But in some cases if you want to create something more, you need to use contracts. So, how do you implement a contract?

It is extremely simple, and one example can illuminate the whole concept. Actually, you have used it already! Suppose you have a controller called `BookController` through which you want to maintain a long list of your favorite books. To do that, you need to store books in a database. To do that, you can bind your `Book` model in your `routes.php` file first, and then using a resource of Book Controller, you can do all kinds of CRUD operations. In doing so, you need to log in.

Consider this code:

```
//code 9.2
public function store(Request $request)
{
  if (Auth::check()) {
    // The user is logged in
  }
}
```

You can check whether the user is logged in, and depending on that, you can add only your favorite books. For this check, you use the `Auth` facade. As long as you don't want a more decoupled state, it works fine. But if you'd rather follow the SOLID principle and want a more decoupled state, then instead of depending on a concrete implementation, you would have to adopt a more robust abstract approach. And in that case, a contract comes to your rescue.

Taylor Otwell himself keeps GitHub repositories on contracts, so you should take a look at that. All of the Laravel contracts live at `https://github.com/illuminate/contracts`.

So, you can either use your `Auth` facade directly in the methods or inject it through the constructor. Understand that when you use a facade directly, it assumes the framework is tightly coupled with a concrete implementation. But considering the SOLID principle, you want a decoupled state. What to do? You can inject your `Auth` facade through a constructor like in the following code and rewrite the code in this way:

```
//code 9.3
use Illuminate\Http\Request;
use App\Http\Requests;
use App\Http\Controllers\Controller;
use Illuminate\Auth\Guard as Auth;
class BooksController extends Controller
{
/**
* Display a listing of the resource
*
* @return Response
*/
protected $auth;
public function __construct(Auth $auth) {
$this-->auth = $auth;
}
public function store(Request $request)
{
$this->auth->attempt();
}
}
```

Now it is much better; you have injected an Auth instance through your constructor. Yes, it is better than before, but still it lacks the SOLID design principle. It depends upon a concrete class like the following:

```
//code 9.4
namespace Illuminate\Auth;
use RuntimeException;
use Illuminate\Support\Str;
use Illuminate\Contracts\Events\Dispatcher;
use Illuminate\Contracts\Auth\UserProvider;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Illuminate\Contracts\Auth\Guard as GuardContract;
use Illuminate\Contracts\Cookie\QueueingFactory as CookieJar;
use Illuminate\Contracts\Auth\Authenticatable as UserContract;
use Symfony\Component\HttpFoundation\Session\SessionInterface;
class Guard implements GuardContract {....}
//code is incomplete for brevity
```

As you can see, when you use this line of code in your `BookController`, you actually inject an instance based on a concrete implementation and not on abstraction.

```
use Illuminate\Auth\Guard as Auth
```

When you unit test your code, you will have to rewrite it. Moreover, whenever you call any methods through this instance, it is aware of your framework.

But you need to make it completely unaware of your framework and become loosely coupled. So, you have to change one line of code in your `BookController`. Instead of using this line of code:

```
use Illuminate\Auth\Guard as Auth;
```

you write this line of code:

```
use Illuminate\Contracts\Auth\Guard as Auth;
```

That is it! Now your `Auth` instance is completely loosely coupled, and you can change this line of code in the `store()` method:

```
if (Auth::check()) {
// The user is logged in
}
```

to this line of code:

```
$this-->auth->attempt();
```

Your application is now more sophisticated because it follows the SOLID design principle and is completely loosely coupled. Finally, if you were to check the interface `Illuminate\Contracts\Auth\Guard`, what would you see? Let's take a look so that you can understand what happens behind the scenes. The code of that interface is pretty big, so for brevity I have cut the `attempt()` method here.

The interface looks like this:

```
//code 9.6
namespace Illuminate\Contracts\Auth;
interface Guard
{
/**
* Attempt to authenticate a user using the
given credentials.
*
* @param array $credentials
* @param bool $remember
* @param bool $login
* @return bool
*/
public function attempt(array $credentials = [],
$remember = false, $login = true);
.....
}
//this code is incomplete for brevity
```

Now your code is not coupled to any vendor or even to Laravel. You are not compelled to follow a strict methodology by using a concrete class. You can simply implement your own, alternative methodology from any contract.

How a Container Works in Laravel

Let's first try to understand what a Laravel service container is. Basically, it is a powerful tool for managing class dependencies and performing dependency injection. The dependencies can be injected via a constructor or through a method. If you can separate all the resources and make them independent, you can use their dependencies through method injection. If you have lots of methods to handle, then you can use the constructor method.

Let's consider an example where you want to get all the users in one place. First, you get the routes, as shown here:

```
//code 9.7
Route::get('allusers', 'UserController@getAllUsers');
```

So, here the method is `getAllUsers()`, and the URI is `allusers`. Now you can make one user repository to get all the users. This will decouple your application. You do not have to hit the database in your `UserController`. You do not even have to use your model in the controller.

You have made a user repository class called `DBUserRepository.php` in your `app/Repositories/DBUserRepositories` directory.

```
//code 9.8
<?php namespace RepositoryDB;

use RepositoryInterface\UserRepositoryInterface as UserRepositoryInterface;

use App\User;
use Illuminate\Http\Request;

class DBUserRepository implements UserRepositoryInterface {

    public function all() {

        return User::all();

    }
}
```

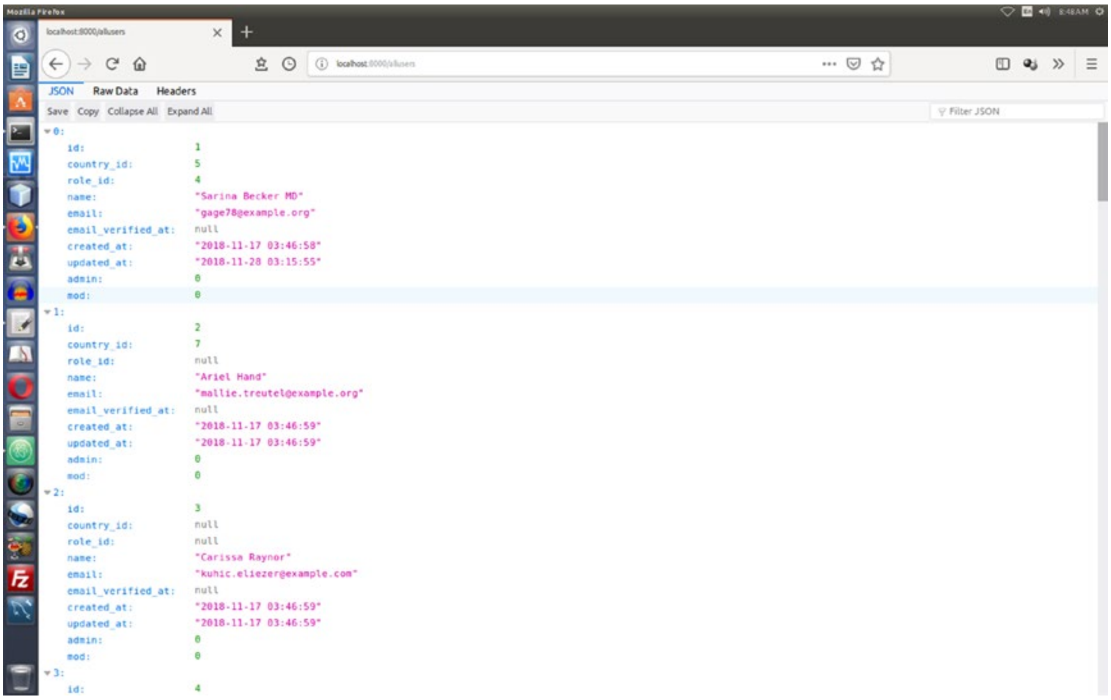
The interface code looks like this:

```
//code 9.9
<?php namespace RepositoryInterface;

 interface UserRepositoryInterface {

     public function all();

}
```

`UserController` now can use either technique, the constructor injection or the method injection.

```
//code 9.10
<?php

namespace App\Http\Controllers;

use App\User;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;

use RepositoryDB\DBUserRepository as DBUserRepository;

class UserController extends Controller {

    public $users;

//constructor injection
    public function __construct(DBUserRepository $users) {

        $this->users = $users;

    }
//method injunction
    public function getAllUsers(DBUserRepository $users){
      return $this->users->all();
    }
//other methods and code continue
}
```

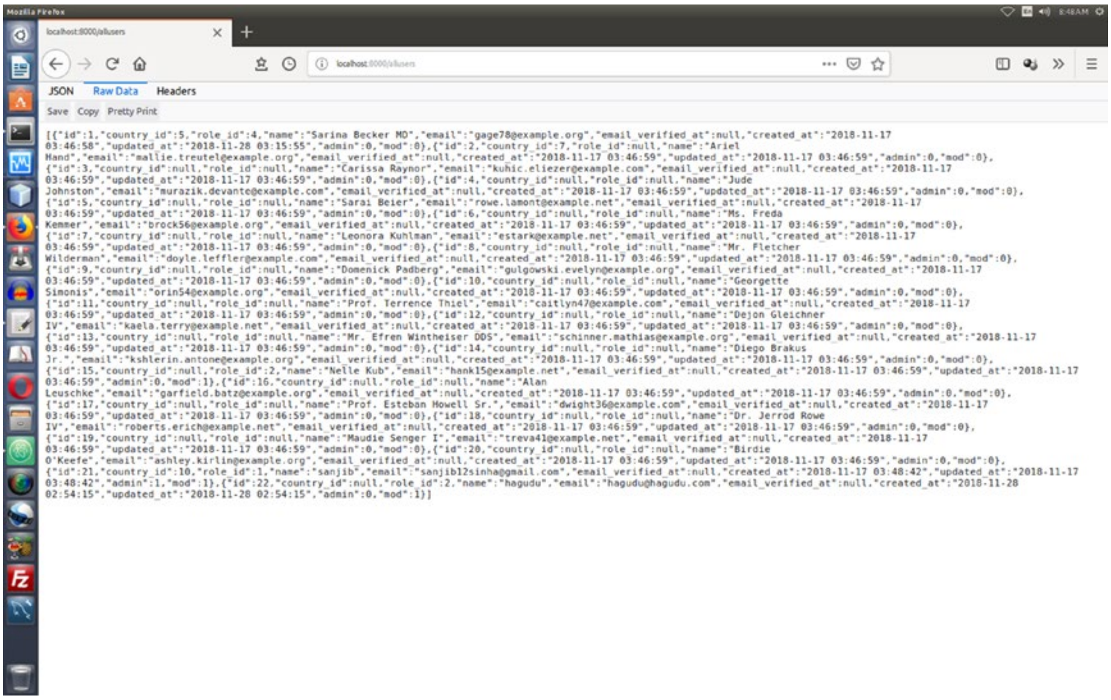
If you hit the URI `http://localhost:8000/allusers`, you first see the JSON output, as shown in Figure 9-1.



```
▾ 0:
    id:                  1
    country_id:          5
    role_id:             4
    name:                "Sarina Becker MD"
    email:               "gage78@example.org"
    email_verified_at:   null
    created_at:          "2018-11-17 03:46:58"
    updated_at:          "2018-11-28 03:15:55"
    admin:               0
    mod:                 0
▾ 1:
    id:                  2
    country_id:          7
    role_id:             null
    name:                "Ariel Hand"
    email:               "mallie.treutel@example.org"
    email_verified_at:   null
    created_at:          "2018-11-17 03:46:59"
    updated_at:          "2018-11-17 03:46:59"
    admin:               0
    mod:                 0
▾ 2:
    id:                  3
    country_id:          null
    role_id:             null
    name:                "Carissa Raynor"
    email:               "kuhic.eliezer@example.com"
    email_verified_at:   null
    created_at:          "2018-11-17 03:46:59"
    updated_at:          "2018-11-17 03:46:59"
    admin:               0
    mod:                 0
▾ 3:
    id:                  4
```

Figure 9-1. *The example of method injection and a JSON output*

You can get a view of the raw data output also, as shown in Figure 9-2.



```
[{"id":1,"country_id":5,"role_id":4,"name":"Sarina Becker MD","email":"gage78@example.org","email_verified_at":null,"created_at":"2018-11-17
03:46:58","updated_at":"2018-11-28 03:15:55","admin":0,"mod":0},{"id":2,"country_id":7,"role_id":null,"name":"Ariel
Hand","email":"mallie.treutel@example.org","email_verified_at":null,"created_at":"2018-11-17 03:46:59","updated_at":"2018-11-17 03:46:59","admin":0,"mod":0},
{"id":3,"country_id":null,"role_id":null,"name":"Carissa Raynor","email":"kuhic.eliezer@example.com","email_verified_at":null,"created_at":"2018-11-17
03:46:59","updated_at":"2018-11-17 03:46:59","admin":0,"mod":0},{"id":4,"country_id":null,"role_id":null,"name":"Jude
Johnston","email":"murazik.devante@example.com","email_verified_at":null,"created_at":"2018-11-17 03:46:59","updated_at":"2018-11-17 03:46:59","admin":0,"mod":0},
{"id":5,"country_id":null,"role_id":null,"name":"Sarai Beier","email":"rowe.lamont@example.net","email_verified_at":null,"created_at":"2018-11-17
03:46:59","updated_at":"2018-11-17 03:46:59","admin":0,"mod":0},{"id":6,"country_id":null,"role_id":null,"name":"Ms. Freda
Kemmer","email":"brock56@example.org","email_verified_at":null,"created_at":"2018-11-17 03:46:59","updated_at":"2018-11-17 03:46:59","admin":0,"mod":0},
{"id":7,"country_id":null,"role_id":null,"name":"Leonora Kuhlman","email":"estark@example.net","email_verified_at":null,"created_at":"2018-11-17
03:46:59","updated_at":"2018-11-17 03:46:59","admin":0,"mod":0},{"id":8,"country_id":null,"role_id":null,"name":"Mr. Fletcher
Wilderman","email":"doyle.leffler@example.com","email_verified_at":null,"created_at":"2018-11-17 03:46:59","updated_at":"2018-11-17 03:46:59","admin":0,"mod":0},
{"id":9,"country_id":null,"role_id":null,"name":"Domenick Padberg","email":"gulgowski.evelyn@example.org","email_verified_at":null,"created_at":"2018-11-17
03:46:59","updated_at":"2018-11-17 03:46:59","admin":0,"mod":0},{"id":10,"country_id":null,"role_id":null,"name":"Georgette
Simonis","email":"orin54@example.org","email_verified_at":null,"created_at":"2018-11-17 03:46:59","updated_at":"2018-11-17 03:46:59","admin":0,"mod":0},
{"id":11,"country_id":null,"role_id":null,"name":"Prof. Terrence Thiel","email":"caitlyn47@example.com","email_verified_at":null,"created_at":"2018-11-17
03:46:59","updated_at":"2018-11-17 03:46:59","admin":0,"mod":0},{"id":12,"country_id":null,"role_id":null,"name":"Dejon Gleichner
IV","email":"kaela.terry@example.net","email_verified_at":null,"created_at":"2018-11-17 03:46:59","updated_at":"2018-11-17 03:46:59","admin":0,"mod":0},
{"id":13,"country_id":null,"role_id":null,"name":"Mr. Efren Wintheiser DDS","email":"schinner.mathias@example.org","email_verified_at":null,"created_at":"2018-11-17
03:46:59","updated_at":"2018-11-17 03:46:59","admin":0,"mod":0},{"id":14,"country_id":null,"role_id":null,"name":"Diego Brakus
Jr.","email":"kshlerin.antone@example.org","email_verified_at":null,"created_at":"2018-11-17 03:46:59","updated_at":"2018-11-17 03:46:59","admin":0,"mod":0},
{"id":15,"country_id":null,"role_id":2,"name":"Nelle Kub","email":"hank15@example.net","email_verified_at":null,"created_at":"2018-11-17 03:46:59","updated_at":"2018-11-17
03:46:59","admin":0,"mod":1},{"id":16,"country_id":null,"role_id":null,"name":"Alan
Leuschke","email":"garfield.batz@example.org","email_verified_at":null,"created_at":"2018-11-17 03:46:59","updated_at":"2018-11-17 03:46:59","admin":0,"mod":0},
{"id":17,"country_id":null,"role_id":null,"name":"Prof. Esteban Howell Sr.","email":"dwight36@example.com","email_verified_at":null,"created_at":"2018-11-17
03:46:59","updated_at":"2018-11-17 03:46:59","admin":0,"mod":0},{"id":18,"country_id":null,"role_id":null,"name":"Dr. Jerrod Rowe
IV","email":"roberts.erich@example.net","email_verified_at":null,"created_at":"2018-11-17 03:46:59","updated_at":"2018-11-17 03:46:59","admin":0,"mod":0},
{"id":19,"country_id":null,"role_id":null,"name":"Maudie Senger I","email":"treva41@example.net","email_verified_at":null,"created_at":"2018-11-17
03:46:59","updated_at":"2018-11-17 03:46:59","admin":0,"mod":0},{"id":20,"country_id":null,"role_id":null,"name":"Birdie
O'Keefe","email":"ashley.kirlin@example.org","email_verified_at":null,"created_at":"2018-11-17 03:46:59","updated_at":"2018-11-17 03:46:59","admin":0,"mod":0},
{"id":21,"country_id":10,"role_id":1,"name":"sanjib","email":"sanjib12sinha@gmail.com","email_verified_at":null,"created_at":"2018-11-17 03:48:42","updated_at":"2018-11-17
03:48:42","admin":1,"mod":1},{"id":22,"country_id":null,"role_id":2,"name":"hagudu","email":"hagudu@hagudu.com","email_verified_at":null,"created_at":"2018-11-28
02:54:15","updated_at":"2018-11-28 02:54:15","admin":0,"mod":1}]
```

***Figure 9-2.** The example of method injection as raw data*

So, you have injected a service that is able to retrieve all users. `UserRepository` gets all users from a database. In the next section, you will see how you can bind your classes and how your service container bindings are registered within the service container.

The output of method injection header is as follows:

```
//output
//method injection Header
Cache-Control: no-cache, private
Connection: close
Content-Type: application/json
Date: Thu, 14 Mar 2019 03:17:16 +0000, Thu, 14 Mar 2019 03:17:16 GMT
Host: localhost:8000
X-Powered-By: PHP/7.2.15-1+ubuntu16.04.1+deb.sury.org+1

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,
image/webp,*/*;q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.5
```

```
Connection: keep-alive
DNT: 1
Host: localhost:8000
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:65.0) Gecko/20100101
Firefox/65.0
```

In the next section, you will see more examples of the relationship between containers and classes and application bindings.

Containers and Classes

The service container is one of Laravel's greatest offerings. You can type-hint a lot of classes and interfaces, and a lot of intelligence is going on inside. Laravel automatically resolves the issues using reflection. In such cases, Laravel passes on instances, recursively bringing about the chained instances.

Consider this code first:

```
//code 9.11
/**
 * Some comments about class Baz
 */

class Baz {};

/**
 * Some comments about class Bax
 */
class Bax
{
  public $baz;
  function __construct(Baz $baz)
  {
    $this->baz = $baz;
  }
}
```

```
 /**
  * Some comments about class Bar
  */
 class Bar
 {
   public $bax;
   function __construct(Bax $bax)
   {
     $this->bax = $bax;
   }
 }
 Route::get('bar', function (Bar $bar) {
     dd($bar);
 });
```

The output is as follows:

```
//output
Bar {#299 ▼
  +bax: Bax {#300 ▼
    +baz: Baz {#301}
  }
}
```

If you want to get the Bax instance, you can do that by tweaking the last part, as shown here:

```
//code 9.12
Route::get('bar', function (Bar $bar) {
     dd($bar->bax);
 });
```

Here is the output:

```
//output
Bax {#300 ▼
  +baz: Baz {#301}
}
```

For beginners, this might look a little confusing. But once you understand it, you will find that it is one of Laravel's greatest offerings.

Why? Let me explain the previous code, and you will understand how Laravel resolves the class dependencies and method injection automatically.

A service container is like a container that houses all of the classes and its bindings. What is important is that you can type-hint any object and get that class instance automatically.

In this code, you have type-hinted the Bar object, and through the Bar instance, you have another class instance, Bax, as shown here:

```
Route::get('bar', function (Bar $bar) {
     dd($bar->bax);
 });
```

When running the code (code 9.11), you get this output:

```
//output of code 9.11
Bar {#299 ▼
  +bax: Bax {#300 ▼
    +baz: Baz {#301}
  }
}
```

In code 9.11, you are type-hinting only the Bar object, but you have chained instances of other class dependencies. So many things are happening here.

When you pass the Bar object through the closure, Laravel finds out that you have already passed the Bax object through the Bar constructor. Moreover, through the Bax constructor, you are passing the Baz object. So, a kind of method chaining of class dependencies takes place. However, Laravel has enough intelligence to resolve these complex issues using reflection automatically.

According to Laravel's documentation, you can also register a binding using the bind method. To do that, you need to pass the class or interface name that you want to register along with a closure that returns an instance of the class.

If you try to do the same thing without trying to bind the interface, an exception is raised.

```
//code 9.13
class Bax implements InterfaceBax {};
```

```
 class Bar
 {
   public $bax;

   function __construct(InterfaceBax $bax)
   {
     $this->bax = $bax;
   }
 }
 Route::get('bar', function (Bar $bar) {
     dd($bar);
 });
```

It gives you the following output:

```
//output
Target [InterfaceBax] is not instantiable while building [Bar].
```

But it works fine when you define the interface properly, and moreover, you can bind it properly in this way:

```
//code 9.14
 interface InterfaceBax
 {
   // code...
 }

class Bax implements InterfaceBax
{

}
 class Bar
 {
   public $bax;

   function __construct(InterfaceBax $bax)
   {
     $this->bax = $bax;
   }
 }
```

```
App::bind('Bar', function(){
  return new Bar(new Bax);
});

 Route::get('bar', function (Bar $bar) {
     dd($bar);
 });
```

Now you get this output:

```
//output
Bar {#295 ▼
  +bax: Bax {#296}
}
```

You can shorten the code in a more intelligent way, like this where you can just bind the interface and the class name:

```
//code 9.15
 interface InterfaceBax
 {
   // code...
 }

class Bax implements InterfaceBax
{

}
 class Bar
 {
   public $bax;

   function __construct(InterfaceBax $bax)
   {
     $this->bax = $bax;
   }
 }
```

```
App::bind('InterfaceBax', 'Bax');

 Route::get('bar', function (Bar $bar) {
     dd($bar);
 });
```

Look at this line in particular:

```
App::bind('InterfaceBax', 'Bax');
```

This changes the whole scenario. Now you can bind it to a container and resolve something out of the container like this:

```
//code 9.16
 interface InterfaceBax
 {
   // code...
 }

class Bax implements InterfaceBax
{

}
 class Bar
 {
   public $bax;

   function __construct(InterfaceBax $bax)
   {
     $this->bax = $bax;
   }
 }

App::bind('InterfaceBax', 'Bax');

 Route::get('bar', function () {
     $bar = App::make('InterfaceBax');
     dd($bar);
 });
```

In the previous code, look at this line:

```
$bar = App::make('InterfaceBax');
```

Here you use the `make()` function to resolve it and get this output straight out of the box:

```
//output
Bax {#288}
```

You could have gotten the same result by using a helper function like `app()`. Look at these three variations of code:

```
//code 9.17
 Route::get('bar', function () {
     $bar = app()->make('InterfaceBax');
     dd($bar);
 });
```

You can also pass the object as an array, as shown here:

```
//code 9.18
Route::get('bar', function () {
     $bar = app()['InterfaceBax'];
     dd($bar);
 });
```

Finally, you can pass it as an argument, as shown here:

```
 Route::get('bar', function () {
     $bar = app('InterfaceBax');
     dd($bar);
 });
```

In these three cases, you get the same result. Remember, the `make` method is used to resolve a class instance out of the container. The `make` method accepts the name of the class or interface you want to resolve.

CHAPTER 10

Working with the Mail Template

Sending mails and notifications through Laravel 5.8 is not complicated. The most popular library to do this with is SwiftMailer (https://swiftmailer.symfony.com/). You can install it in your root application directory.

Local Development

To check your e-mail verification property locally, let's install a fresh version of Laravel in your home/code directory. You need a fresh Laravel installation to start with the concept that the user is not verified initially.

```
//code 10.1
$ composer create-project --prefer-dist laravel/laravel laranew
```

Let's check the version.

```
//code 10.2
ss@ss-H81M-S1:~/code/laranew$ php artisan -V
Laravel Framework 5.8.4
```

This specifies that I have a fresh Laravel 5.8.4 installation in home/code. I have named it laranew.

© Sanjib Sinha 2019
S. Sinha, *Beginning Laravel*, https://doi.org/10.1007/978-1-4842-4991-8_10

Before moving on to explain the e-mail verification process in Laravel 5.8, let's take a look at the code of the User model. You have already learned that a fresh Laravel installation comes with a User model. Specifically, it comes with the following code:

```
//code 10.3
//app/User.php
<?php

namespace App;

use Illuminate\Notifications\Notifiable;
use Illuminate\Contracts\Auth\MustVerifyEmail;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable implements MustVerifyEmail
{
    use Notifiable;

    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = [
        'name', 'email', 'password',
    ];

    /**
     * The attributes that should be hidden for arrays.
     *
     * @var array
     */
    protected $hidden = [
        'password', 'remember_token',
    ];
    /**
     * The attributes that should be cast to native types.
     *
     * @var array
     */
    protected $casts = [
        'email_verified_at' => 'datetime',
    ];
}
```

In the previous code, you are interested in two parts. The first one is as follows:

```
use Illuminate\Contracts\Auth\MustVerifyEmail;
```

The second one is as follows:

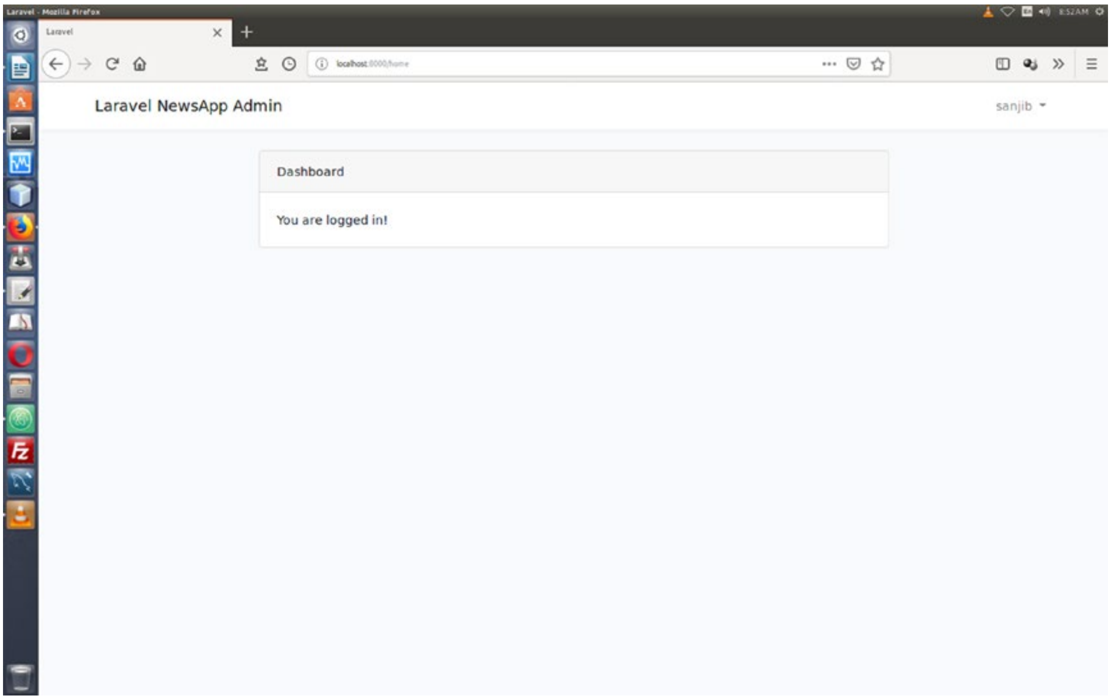
```
protected $casts = [
        'email_verified_at' => 'datetime',
    ];
```

The first line tells you about a contract. You have already learned about the role of a contract. `MustVerifyEmail` has three methods defined in the source code. This contract takes care of one property of user registration: the user must verify their e-mail if you implement the contract in the `User` model.

Let's again refer to the `User` model, as shown here:

```
//code 10.4
class User extends Authenticatable implements MustVerifyEmail
```

In previous chapters when you registered users, you did not implement this contract; so, under normal circumstances, a user is greeted with the screen shown in Figure 10-1.


Laravel NewsApp Admin

sanjib

Dashboard

You are logged in!

Figure 10-1. *The user is greeted with this screen after registration*

Notice that there is no e-mail verification message. This is because you have not implemented that contract in the `User` model. There is another reason also.

To understand this, you can check the `database/migrations/user` table, as shown here:

```
//code 10.5
public function up()
    {
        Schema::create('users', function (Blueprint $table) {
            $table->bigIncrements('id');
            $table->string('name');
            $table->string('email')->unique();
            $table->timestamp('email_verified_at')->nullable();
            $table->string('password');
            $table->rememberToken();
            $table->timestamps();
        });
    }
```

Take a look at this line:

```
$table->timestamp('email_verified_at')->nullable();
```

and compare it with the line I have highlighted in your `User` model, as shown here:

```
protected $casts = [
        'email_verified_at' => 'datetime',
    ];
```

This attribute has been cast to the native type `datetime`, and for that reason in the `user` table Laravel has kept it as `nullable timestamp`.

Because of this, when you register a new user without implementing the `MustVerifyEmail` contract, your database user table shows `null` in that row. Let's check it.

```
//code 10.6
mysql> show databases;
+------------------------+
| Database               |
+------------------------+
| information_schema     |
| b2a                    |
| firstnews              |
| imagery                |
| laravel55              |
| laravelforartisans     |
| laravelforbeginning    |
| laravelmodelrelations  |
| laravelrelations       |
| laravelstarttofinish   |
| myappo                 |
| mymvc                  |
| mysql                  |
| newdata                |
| news                   |
| performance_schema     |
| practiceone            |
| prisma                 |
| sys                   |
| test                  |
| twoprac               |
+-----------------------+
21 rows in set (0.20 sec)

mysql> use newdata;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables;
+------------------+
| Tables_in_newdata |
+------------------+
| migrations        |
| password_resets   |
| users             |
+------------------+
3 rows in set (0.00 sec)

mysql> select * from users;
+----+--------+-------------------------+---------------------+----------
----------------------------------------------------+--------------+----
-----------------+---------------------+
| id | name   | email                   | email_verified_at   | password
| remember_token | created_at          | updated_at          |
+----+--------+-------------------------+---------------------+----------
----------------------------------------------------+--------------+----
-----------------+---------------------+
|  1 | ss     | s@s.com                 | NULL                |
$2y$10$T3lZcOOCC5C/OIRpOSO2SefpKfhtJF8myWaLeLAgNU4nthruQ2Dgu | NULL
| 2019-03-17 03:51:29 | 2019-03-17 03:51:29 |
|  2 | sanjib | sanjib12sinha@gmail.com | 2019-03-17 03:59:08 |
$2y$10$kv34GZLUvIZLt/UqprrjCuOeS22.dI9iOR73vAX5IVfdLOiDpDauu | NULL
| 2019-03-17 03:54:22 | 2019-03-17 03:59:08 |
+----+--------+-------------------------+---------------------+-----------
---------------------------------------------------+---------------+----
-----------------+---------------------+
2 rows in set (0.01 sec)
```

Using Tinker to Find the Verified E-mail

For more clarity, let's use `tinker` and see the first user who has not e-mail verification.

```
//code 10.7
ss@ss-H81M-S1:~/code/laranew$ php artisan tinker
Psy Shell v0.9.9 (PHP 7.2.15-1+ubuntu16.04.1+deb.sury.org+1 — cli) by
Justin Hileman
>>> $user = new App\User;
=> App\User {#2925}
>>> $user;
=> App\User {#2925}
>>> $user = App\User::find(1);
=> App\User {#2933
     id: 1,
     name: "ss",
     email: "s@s.com",
     email_verified_at: null,
     created_at: "2019-03-17 03:51:29",
     updated_at: "2019-03-17 03:51:29",
   }
>>>
```

Take a look at this line from the previous code:

```
email_verified_at: null,
```

It is `null` because you have kept it as `null`. Moreover, you have not verified the e-mail.

In the next step, you will implement e-mail verification so you can see how it works with the local Laravel installation. You can test it with the help of your log file.

Before that, all you need to do is change the code of the env file in this manner:

```
//code 10.8
MAIL_DRIVER=log
MAIL_HOST=smtp.Mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=null
MAIL_PASSWORD=null
MAIL_ENCRYPTION=null
```

You have changed the `MAIL_DRIVER` property from `smtp` to `log`.

Next, take a look at this part of the `app/Http/Kernel.php` file:

```
//code 10.9
//app/Http/Kernel.php
protected $routeMiddleware = [
        'auth' => \App\Http\Middleware\Authenticate::class,
        'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasic
        Auth::class,
        'bindings' => \Illuminate\Routing\Middleware\SubstituteBindings::class,
        'cache.headers' => \Illuminate\Http\Middleware\SetCacheHeaders::class,
        'can' => \Illuminate\Auth\Middleware\Authorize::class,
        'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
        'signed' => \Illuminate\Routing\Middleware\ValidateSignature::class,
        'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
        'verified' => \Illuminate\Auth\Middleware\EnsureEmailIsVerified::class,
    ];
```

The last line from the previous code is important here.

```
'verified' => \Illuminate\Auth\Middleware\EnsureEmailIsVerified::class
```

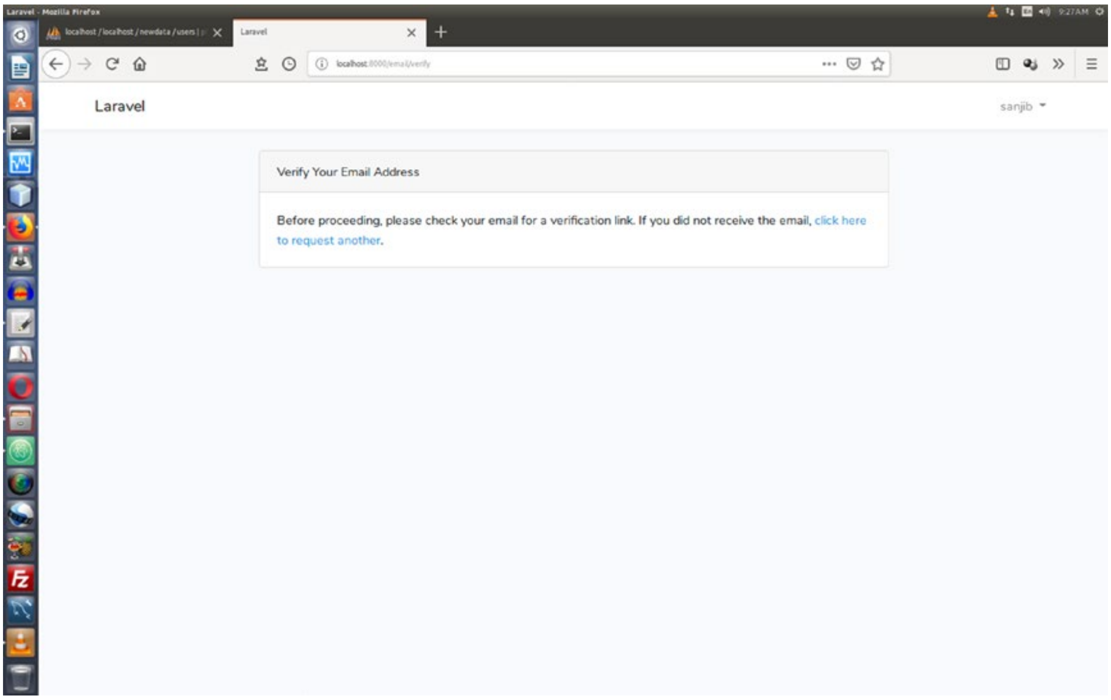
Either you can assign these route middleware components to groups or you can use them individually. For this e-mail verification process, you will use the last middleware individually.

Changing the Route

To use this middleware component, you need to change your route `web.php` file in this way:

```
//code 10.10
//routes/web.php
Route::get('/', function () {
    return view('welcome');
})->middleware('verified');
Auth::routes(['verify' => true]);
```

You have used the e-mail verification middleware at the right place; now you can register a new user to see what happens, as shown in Figure 10-2.



Verify Your Email Address

Before proceeding, please check your email for a verification link. If you did not receive the email, click here to request another.

Figure 10-2. *E-mail verification notice has been issued*

Next, go to `storage/logs/laravel-2019-03-17.log` to see the following output:

```
//code 10.11
// storage/logs/laravel-2019-03-17.log
Content-Type: text/plain; charset=utf-8
Content-Transfer-Encoding: quoted-printable
```

```
[Laravel](http://localhost)

# Hello!

Please click the button below to verify your email address.

Verify Email Address: http://localhost:8000/email/verify/2?expires=1552798
462&signature=80dd1cd1315533a03d03f66fb0e90b5a9b21c454257b6a7e96b4d5ed9059
b2f1

If you did not create an account, no further action is required.

Regards,Laravel

If you're having trouble clicking the "Verify Email Address" button, copy
and paste the URL below into your web browser:
[http://localhost:8000/email/verify/2?expires=1552798462&signature=80d
d1cd1315533a03d03f66fb0e90b5a9b21c454257b6a7e96b4d5ed9059b2f1](http://
localhost:8000/email/verify/2?expires=1552798462&signature=80dd1cd1315533a0
3d03f66fb0e90b5a9b21c454257b6a7e96b4d5ed9059b2f1)

© 2019 Laravel. All rights reserved.
```

Basically, this sends an e-mail to the newly registered user and gives the user links that they can click to verify their e-mail.

So. let's click this link and see what happens:

```
http://localhost:8000/email/verify/2?expires=1552798462&signature=80dd1cd13
15533a03d03f66fb0e90b5a9b21c454257b6a7e96b4d5ed9059b2f1
```

Next, open your terminal. By using Tinker, you can see the status of the second user.

```
//code 10.12
ss@ss-H81M-S1:~$ cd code/laranew/
ss@ss-H81M-S1:~/code/laranew$ php artisan tinker
Psy Shell v0.9.9 (PHP 7.2.15-1+ubuntu16.04.1+deb.sury.org+1 — cli) by
Justin Hileman
>>> $user = App\User::find(2);
=> App\User {#2932
     id: 2,
     name: "sanjib",
    email: "sanjib12sinha@gmail.com",
    email_verified_at: "2019-03-17 03:59:08",
    created_at: "2019-03-17 03:54:22",
    updated_at: "2019-03-17 03:59:08",
  }
>>>
```

Look at this line:

```
email_verified_at: "2019-03-17 03:59:08",
```

The e-mail has been verified. It no longer shows as `null`.

Now if you want to know what happens in the background, you can take a look at the event listener property of the file `app/Providers/EventServiceProvider.php`.

```
//code 10.13
// app/Providers/EventServiceProvider.php
protected $listen = [
        Registered::class => [
            SendEmailVerificationNotification::class,
        ],
    ];
```

Whenever you implement this attribute to the `User` model (`class User extends Authenticatable implements MustVerifyEmail`), this event fires when a new user registers.

Sending E-mail and Notifications

In this section, you will see how to send e-mails using a Laravel application. You will do this through a live web portal at `https://sanjib.site`, although the same test can be done in the local environment.

Then you will learn how to send notifications. I have done that locally. In both cases, I have used the free Mailtrap service. Laravel also has the option that you can buy third-party e-mail services.

Sending E-mails

There are several options for sending e-mails from your Laravel 5.8 application. The default option given in your .env file is SMTP, but you can override it with several other options, such as Mailgun, SparkPost, Amazon SES, PHP's mail function, and sendmail. Note that sending mail through a cloud-based service requires registration and a service cost.

Although the API-based services such as Mailgun and Sparkpost are simpler and faster than the default SMTP servers, they require the Guzzle HTTP library. You can install it via the Composer package manager using this command:

```
//code 10.14
$ composer require guzzlehttp/guzzle
```

After the installation of Guzzle, you need to change the driver option in your config/mail.php configuration file to mailgun.

This config/mail.php file is extremely important for your e-mail service. You will see that code in a minute.

Here, I am going to show you the mail-sending process using Mailtrap. It uses the default SMTP driver, and it is free; you can use a dummy mailbox to test the process and actually inspect the final e-mails in Mailtrap's message viewer.

```
//code 10.15
//config/mail.php
<?php

return [

    /*
    |--------------------------------------------------------------------------
    | Mail Driver
    |--------------------------------------------------------------------------
    |
    | Laravel supports both SMTP and PHP's "mail" function as drivers for the
    | sending of e-mail. You may specify which one you're using throughout
    | your application here. By default, Laravel is setup for SMTP mail.
    |
| Supported: "smtp", "sendmail", "mailgun", "mandrill", "ses",
|            "sparkpost", "log", "array"
|
*/

'driver' => env('MAIL_DRIVER', 'smtp'),

/*
|----------------------------------------------------------------------
| SMTP Host Address
|----------------------------------------------------------------------
|
| Here you may provide the host address of the SMTP server used by your
| applications. A default option is provided that is compatible with
| the Mailgun mail service which will provide reliable deliveries.
|
*/

'host' => env('MAIL_HOST', 'smtp.Mailtrap.io'),

/*
|----------------------------------------------------------------------
| SMTP Host Port
|----------------------------------------------------------------------
|
| This is the SMTP port used by your application to deliver e-mails to
| users of the application. Like the host we have set this value to
| stay compatible with the Mailgun e-mail application by default.
|
*/

'port' => env('MAIL_PORT', 465),

/*
|----------------------------------------------------------------------
| Global "From" Address
|----------------------------------------------------------------------
|
    | You may wish for all e-mails sent by your application to be sent from
    | the same address. Here, you may specify a name and address that is
    | used globally for all e-mails that are sent by your application.
    |
    */

    'from' => [
        'address' => env('MAIL_FROM_ADDRESS', 'me@sanjib.site'),
        'name' => env('MAIL_FROM_NAME', 'Sanjib Sinha'),
    ],

    /*
    |--------------------------------------------------------------------
    | E-Mail Encryption Protocol
    |--------------------------------------------------------------------
    |
    | Here you may specify the encryption protocol that should be used when
    | the application send e-mail messages. A sensible default using the
    | transport layer security protocol should provide great security.
    |
    */

    'encryption' => env('MAIL_ENCRYPTION', 'tls'),

    /*
    |--------------------------------------------------------------------
    | SMTP Server Username
    |--------------------------------------------------------------------
    |
    | If your SMTP server requires a username for authentication, you should
    | set it here. This will get used to authenticate with your server on
    | connection. You may also set the "password" value below this one.
    |
    */
    'username' => env('MAIL_USERNAME'),

    'password' => env('MAIL_PASSWORD'),

    /*
    |----------------------------------------------------------------------
    | Sendmail System Path
```

These lines are especially important to note in the previous code:

```
'driver' => env('MAIL_DRIVER', 'smtp'),
'host' => env('MAIL_HOST', 'smtp.Mailtrap.io'),
'port' => env('MAIL_PORT', 465),
'from' => [
        'address' => env('MAIL_FROM_ADDRESS', 'me@sanjib.site'),
        'name' => env('MAIL_FROM_NAME', 'Sanjib Sinha'),
    ],
```

In the `from` field, you can use any other e-mail address and your name. Before using the Mailtrap service, all you need to do is to register with Mailtrap.

Next, the `.env` file is important. You need to change this part according to the username and password of your Mailtrap service.

```
//code 10.16
//.env
MAIL_DRIVER=smtp
MAIL_HOST=smtp.Mailtrap.io
MAIL_PORT=465
MAIL_USERNAME=**************
MAIL_PASSWORD=**************
MAIL_ENCRYPTION=null
```

This time I am going to send mail from my live web application at `https://sanjib.site`. Don't worry; you can do the same thing from your local Laravel application using `http://localhost:8000`.

First, you configure the file `config/mail.php`; second, you change the `.env` file with your Mailtrap username and password. Remember one thing: when you are testing your application online, you need to use port 465. When you are doing the same thing, use port 2525 that comes by default with your installed Laravel application.

The next step will take you to the `artisan` command. If you are online, use SSH, and if you are in a local environment, use your terminal.

```
//code 10.17
$ php artisan make:mail SendMailable
```

This command creates a `Mail` folder inside the `app` directory, and in the `Mail` folder a new `SendMailable` class has been created.

```
//code 10.18
//app/Mail/SendMailable.php
<?php

namespace App\Mail;

use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendMailable extends Mailable
{
    use Queueable, SerializesModels;
    public $name;

    /**
     * Create a new message instance.
     *
     * @return void
     */
    public function __construct($name)
    {
        $this->name = $name;
    }

    /**
     * Build the message.
     *
     * @return $this
     */
    public function build()
    {
        return $this->view('email.name');
    }
}
```

By default the SendMailable class will instantiate a message. For the sake of brevity, I have kept it simple and used a name attribute. This name variable will be passed to the following build method automatically, which will return a view Blade page. This means you can pass a full HTML page with your message to the recipient.

So, you can take the next step and create a simple view page called name.blade.php inside resources/views/email.

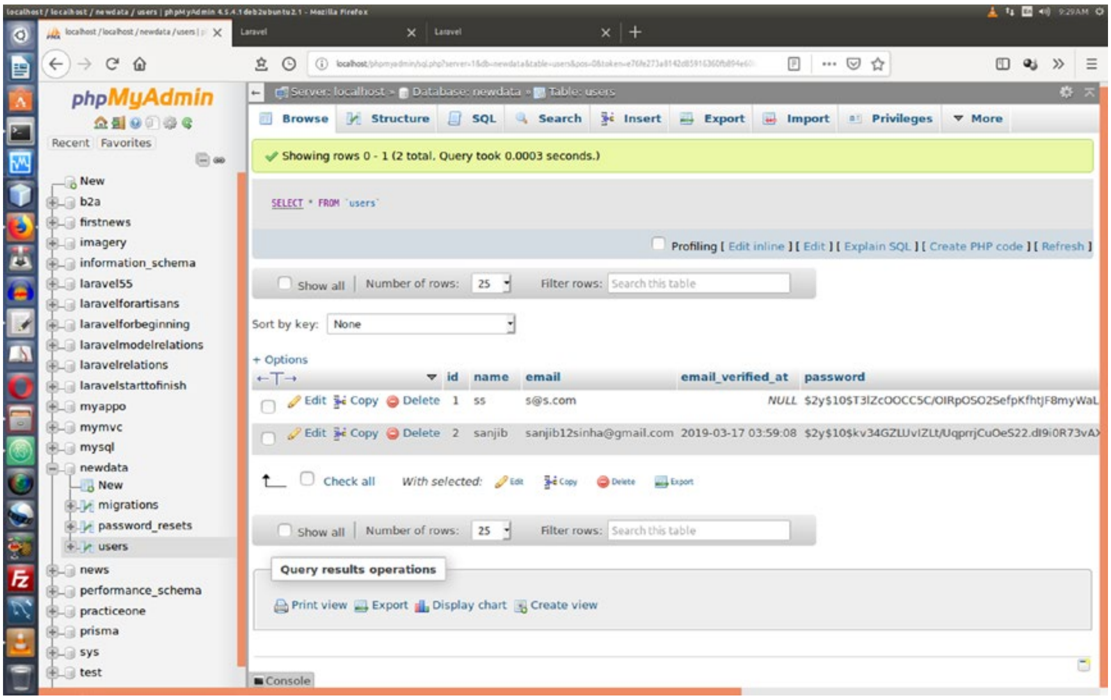
```
//code 10.19
//resources/views/email/name.blade.php
<div>
    Hi, This is : {{ $name }}
</div>
```

At the same time in your HomeController, you will create a new method named mail.

```
//code 10.20
//app/Http/Controllers/HomeController.php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use Illuminate\Support\Facades\Mail;
use App\Mail\SendMailable;

class HomeController extends Controller
{
    /**
     * Create a new controller instance.
     *
     * @return void
     */
    public function __construct()
    {
        $this->middleware('auth');
    }

    /**
     * Show the application dashboard.
     *
     * @return \Illuminate\Contracts\Support\Renderable
     */
    public function index()
    {
        return view('home');
    }

   public function mail()
{
   $name = 'Sanjib';
   Mail::to('me@sanjib.site')->send(new SendMailable($name));

   return 'Email was sent';
}
}
```

Since I am going to test the mail-sending process online, I have chosen the `HomeController` so that it will not work without authentication.

In your routes, you need to register the method now so that when you type that URI, the user will be verified and you will see them in your PHPMyAdmin interface, as shown in Figure 10-3.



✓ Showing rows 0 - 1 (2 total, Query took 0.0003 seconds.)

```
SELECT * FROM `users`
```

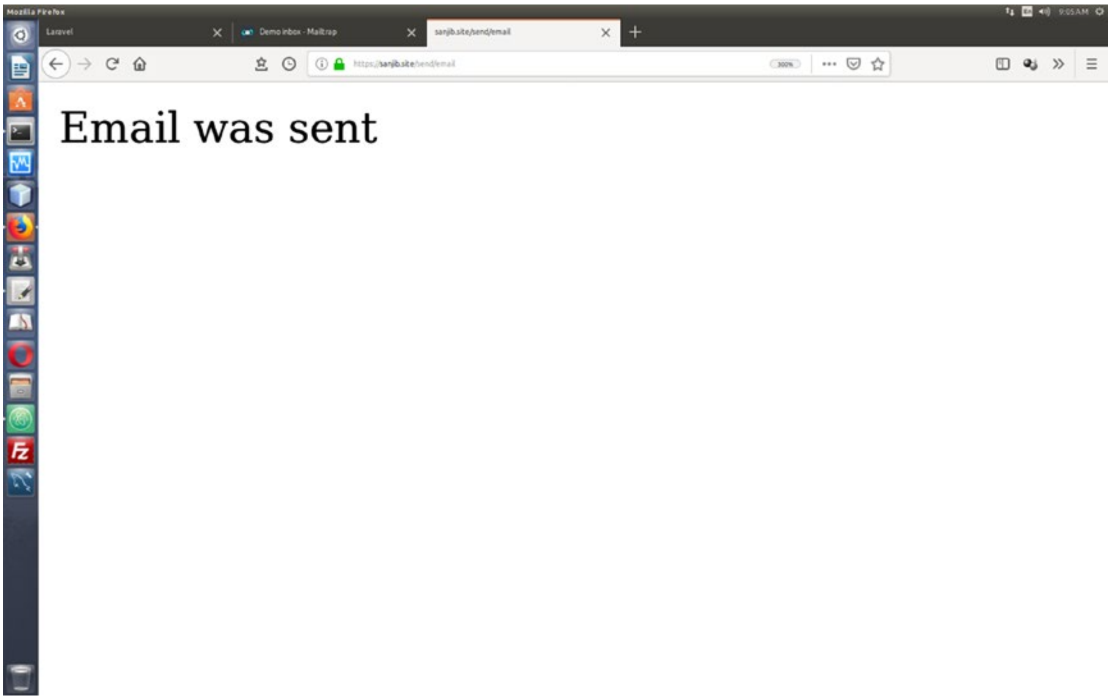
id	name	email	email_verified_at	password
1	ss	s@s.com	NULL	$2y$10$T3lZcOOCC5C/OIRpOSO2SefpKfhtjF8myWaL
2	sanjib	sanjib12sinha@gmail.com	2019-03-17 03:59:08	$2y$10$kv34GZLUvIZLt/UqprrjCuOeS22.dI9i0R73vAX

Figure 10-3. *The user verification is reflected in PHPMyAdmin*

```
//code 10.21
//routes/web.php
Route::get('/send/email', 'HomeController@mail');
```

You have completed all the steps. Now it is time to test whether your mail has been sent successfully.

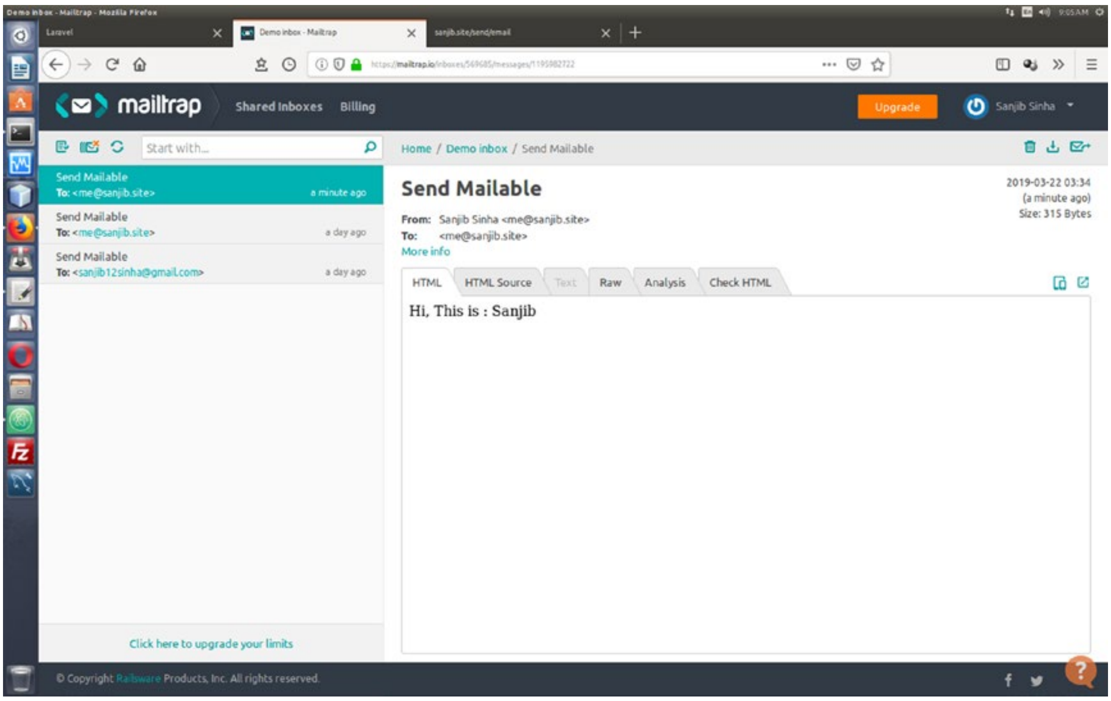
I have typed `https://sanjib.site/send/email`, and I got the response shown in Figure 10-4 in my browser.



Email was sent

Figure 10-4. *The email has been sent successfully*

The next step is to check the Mailtrap inbox, so log in to Mailtrap. You will find that the e-mail was sent successfully "a minute ago" (Figure 10-5).



Send Mailable
To: <me@sanjib.site> a minute ago

Send Mailable
To: <me@sanjib.site> a day ago

Send Mailable
To: <sanjib12sinha@gmail.com> a day ago

Send Mailable

From: Sanjib Sinha <me@sanjib.site>
To: <me@sanjib.site>
More info

2019-03-22 03:34
(a minute ago)
Size: 315 Bytes

Hi, This is : Sanjib

© Copyright Railsware Products, Inc. All rights reserved.

***Figure 10-5.** Mailtrap inbox showing that the Laravel application has sent the email successfully*

Let's take a look at the HTML source. I have used a simple HTML template page, and in your Mailtrap inbox you can see everything, including the HTML source and the raw output; you can also analyze the whole process.

The HTML source is the same as you saw in your `resources/views/email/name.blade.php` file.

```
//code 10.22
<div>
    Hi, This is : Sanjib
</div>
```

The only change is that it has caught the variable $name. You have used the same name variable in your HomeController mail method, as shown here:

```
//code 10.23
   public function mail()
{
   $name = 'Sanjib';
   Mail::to('me@sanjib.site')->send(new SendMailable($name));
   return 'Email was sent';
}
```

The Mailtrap raw output looks like this:

```
//code 10.24
Message-ID: <e792cc1223e458eccf9e01f7620c6be1@sanjib.site>
Date: Fri, 22 Mar 2019 03:34:08 +0000
Subject: Send Mailable
From: Sanjib Sinha <me@sanjib.site>
To: me@sanjib.site
MIME-Version: 1.0
Content-Type: text/html; charset=utf-8
Content-Transfer-Encoding: quoted-printable

<div>
    Hi, This is : Sanjib
</div>
```

Like sending e-mails, sending notifications is easy through any Laravel application. In the next section, you will learn how to test the notification process locally.

How to Send Notifications

You saw how Laravel makes sending e-mails easy. In addition, Laravel provides support for sending notifications. There are varieties of delivery channels you can use while sending a short notification.

You can send the notification through mail or through SMS (via Nexmo, https://www.nexmo.com/). By the way, Nexmo uses APIs for SMS, voice, and phone verifications.

The notifications can also be stored in a database so that you can display them in any web interface.

As you know, any notification should be short and informational in nature. When a new user registers, you may want to get a notification.

Here you will use the mail delivery system using Mailtrap, as you have used it already. First, you need to change the `.env` code to this:

```
//code 10.25
//.env
MAIL_DRIVER=smtp
MAIL_HOST=smtp.Mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=*****************
MAIL_PASSWORD=*****************
MAIL_ENCRYPTION=tls
Next, we will issue this command on your terminal.
//code 10.26
$ php artisan make:notification NewuserRegistered
```

Once you have issued this command, in your `app/Notifications` directory, you get the `NewuserRegistered.php` file.

```
//code 10.27
//app/Notifications/NewuserRegistered.php
<?php

namespace App\Notifications;

use Illuminate\Bus\Queueable;
use Illuminate\Notifications\Notification;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Notifications\Messages\MailMessage;

class NewuserRegistered extends Notification
{
    use Queueable;
    /**
     * Create a new notification instance.
     *
     * @return void
     */
    public function __construct()
    {
        //
    }

    /**
     * Get the notification's delivery channels.
     *
     * @param  mixed  $notifiable
     * @return array
     */
    public function via($notifiable)
    {
        return ['mail'];
    }

    /**
     * Get the mail representation of the notification.
     *
     * @param  mixed  $notifiable
     * @return \Illuminate\Notifications\Messages\MailMessage
     */
    public function toMail($notifiable)
    {
        return (new MailMessage)
                    ->line('A new user just registered.')
                    ->action('MyApp', url('/'))
                    ->line('Thank you for using our application!');
    }
    /**
     * Get the array representation of the notification.
     *
     * @param  mixed  $notifiable
     * @return array
     */
    public function toArray($notifiable)
    {
        return [
            //
        ];
    }
}
```

The default delivery channel of notifications is `mail`. You have not changed that. You can change this part:

```
    public function toMail($notifiable)
    {
        return (new MailMessage)
                    ->line('A new user just registered.')
                    ->action('MyApp', url('/'))
                    ->line('Thank you for using our application!');
    }
```

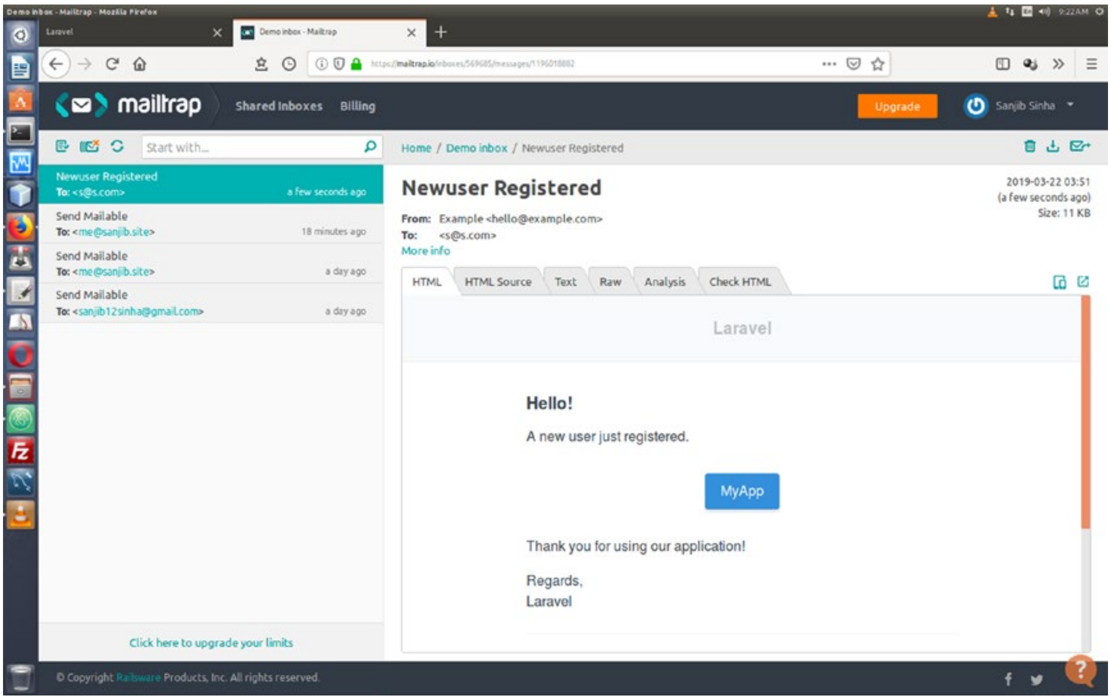
The correct setting depends on what type of notification you want to get or send to the user.

Suppose user 1 is the administrator here; in that case, you want to send the new user registration notification to user 1. So, you need to register the route in this way:

```
//code 10.28
//resources/web.php
use App\Notifications\NewuserRegistered;
use App\User;
Route::get('/notify', function () {
User::find(1)->notify(new NewuserRegistered);
    return view('notify');
});
```

Since I hard-coded everything to give you an idea of how it works, you should type `http://localhost:8000/notify` in your browser to fire the mail.

This automatically sends the notification to your Mailtrap inbox, as shown in Figure 10-6.



Newuser Registered
To: <s@s.com> a few seconds ago

Send Mailable
To: <me@sanjib.site> 18 minutes ago

Send Mailable
To: <me@sanjib.site> a day ago

Send Mailable
To: <sanjib12sinha@gmail.com> a day ago

Newuser Registered

2019-03-22 03:51
(a few seconds ago)
Size: 11 KB

From: Example <hello@example.com>
To: <s@s.com>
More info

Laravel

Hello!

A new user just registered.

Thank you for using our application!

Regards,
Laravel

© Copyright Railsware Products, Inc. All rights reserved.

Figure 10-6. *Notification of new user registration in the inbox of Mailtrap*

Here is the text output of your new user registration notification in your Mailtrap inbox:

```
//code 10.29
[Laravel](http://localhost)

# Hello!

A new user just registered.

MyApp: http://localhost:8000

Thank you for using our application!

Regards,Laravel
```

```
If you're having trouble clicking the "MyApp" button, copy and paste the
URL below
into your web browser: [http://localhost:8000](http://localhost:8000)

© 2019 Laravel. All rights reserved.
```

CHAPTER 11

Events and Broadcasting

In the previous chapter, you learned how to send e-mails and notifications. In this chapter, you will learn about events and broadcasting, which, at the beginning, may look similar to notifications but are not.

What Are Events and Broadcasting?

Although the difference between notification and events may not be clear at first, events and notifications have distinct characteristics and are used for different purposes.

To express this difference in one line, you could say that events are for when you need to do something, and notifications are for when something happens in your application. In other words, events trigger actions (such as when system failure and restoration actions need to be performed), and notifications are sent for all events.

So, events are like announcements of something that you listen for, and notifications are issued when something just happened.

I would like to reiterate that, at the beginning, the difference is so marginal and looks almost invisible that people often use events when notifications are more useful, and vice versa. However, the range of activities that an event can handle is enormous. Because of that, notifications can be managed through events.

Let me clarify with an example.

Suppose, in the administrative panel, you want to know when a new user registers. You do not want to reload the page to get the update but rather have the event carry out the action for you. Although you might think to use a notification, actually here an event could be more useful, as you can do many tasks with the help of one event. Sometimes doing many tasks with the help of a single resource controller can lead you to tightly coupled classes, which is not your goal. In these cases, you could use Laravel Echo, which is an integral property of events and broadcasting. Laravel Echo will automatically append the newly registered users to your list, and you can use the `ShouldBroadcast` interface to sync data with your administrative panel.

© Sanjib Sinha 2019
S. Sinha, *Beginning Laravel*, https://doi.org/10.1007/978-1-4842-4991-8_11

As another example, on any social media platform, each time you log in, you see some notifications about things that happened while you were away. Live notifications also take place when you are present on social media platforms. So, you can use the `notify` method on your Eloquent models such as `App\User` to send an SMS to the newly registered user that their registration has been approved.

Moreover, you can handle notifications through your events, but you cannot do the opposite. On an e-commerce site, as an application developer, you need to manage a lot of steps when a product is purchased. If you want to arrange them in a single controller, it could get messy. On the other hand, it could be a good approach to manage them through an event where you can have different listeners, and through events you can broadcast them to those listeners. These listeners could be the different steps through which you can manage sending notifications, generating invoices, and so on.

So in this application, you can have several listeners through which you can create notifications, send e-mails, and so on.

I hope you understand the slight differences between these two major features of Laravel. Next, you will see how you can set up and configure events and broadcasting.

Setting Up Events and Broadcasting

There is a popular software design pattern called the observer pattern. In the observer pattern, the system is supposed to fire events and broadcast them to listeners, and at the same time different tasks are delegated. You can define listeners who listen to these events. The main advantage of the observer pattern is it decouples the classes.

The concept of events in Laravel is based on this popular software design pattern. It's a really useful feature in a way that allows you to decouple components in a system that otherwise would have resulted in tightly coupled code. It actually adheres to the SOLID design principle: one class, one single task, and one responsibility. The simple observer implementation allows you to subscribe and listen for various events that occur in the application.

As you progress, you will find that there is a small difference between subscribers and listeners. When an event fires, a subscriber can subscribe to multiple event listeners in a single place, whereas a single listener can listen to a single event. Here context is the key. If you want to contain many events in a single place, a subscriber is a good option. However, to understand when you need subscribers, you need to understand the listeners first.

The event classes are stored in the `app/Events` directory, and the listeners are stored in the `app/Listeners` directory.

They can be manually created, or they can be autogenerated. If you want to autogenerate events and listeners, you need to have registered them with the event service provider first.

You will see examples of this in a minute. You can create them using `artisan` commands.

The greatest advantage of events is since they have multiple listeners, the classes are left with single and precise tasks. The listeners do not depend on each other. In the coming sections, you will see how the creation of a message fires events that do several things at the same time. All these tasks can be done through the controller methods `create` and `store` (although, in the `store` method, you have only one event). That is the magic you are going to create.

Creating Events

Creating a simple message creation application will help you to understand the core concept of events and broadcasting. Each time one user posts a message, a single event fires, and the event broadcasts the message to different decoupled listeners.

This event handles multiple loosely coupled listeners that have no interaction with each other. One listener sends an e-mail. Two other listeners handle different tasks, such as sending e-mail messages or notifications in the browser.

Usually in such cases, a JavaScript framework like Vue.js is of great help. You can do splendid things using Vue, and the real-time message handling API Pusher is also helpful. Laravel has boundless support for these two tools.

Let's first start with a resourceful controller, as shown here:

```
//code 11.1
$ php artisan make:controller CreatemessageController -resource
```

For displaying and showing messages, the full controller methods look like this:

```
//code 11.2
//app/Http/Controllers/ CreatemessageController.php
<?php

namespace App\Http\Controllers;
```

```
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;
use App\User;
use App\Message;
use App\Events\AboutTheUser;

class CreatemessageController extends Controller
{

  /**
   * Create a new controller instance.
   *
   * @return void
   */
  public function __construct()
  {
      $this->middleware('auth');
  }
    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
      if (Auth::check()){
        $messages = Message::get();
        return view('messages.index', compact('messages'));
      }
    }

    /**
     * Show the form for creating a new resource.
     *
     * @return \Illuminate\Http\Response
     */
```

```
public function create()
{
  if( Auth::check() ){
      return view('messages.create');
  }
    return view('auth.login');

}

/**
 * Store a newly created resource in storage.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return \Illuminate\Http\Response
 */
public function store(Request $request)
{
    if(Auth::check()){
        $name = Auth::user();
        $message = Message::create([
            'user_id' => Auth::user()->id,
            'message' => $request->input('message')

        ]);
        if($name){

            //step one: fire the event
            event(New AboutTheUser($name));

        }


    }
  }

/**
 * Display the specified resource.
 *
 * @param  int  $id
```

```
     * @return \Illuminate\Http\Response
     */
     public function show(Message $message)
     {
       if( Auth::check() ){
             $message = Message::find($message->id);
             return view('messages.show', ['message' => $message]);
       }
         return view('auth.login');

     }
}
```

As you can see, you don't have any `edit`, `update`, or `destroy` methods here. You are going to create a message, and at the same time it will fire an event like this:

```
//step one: fire the event
event(New AboutTheUser($name));
```

In your `CreatemessageController` controller action `App\Http\Controllers\CreatemessageController@store`, you have only one line of event code, and that event has delegated those tasks to the various listeners.

This is the beauty of Laravel events, where through a single event you can delegate several tasks to different listeners that are loosely coupled and they handle single tasks. You will see how to create events and listeners in a minute; before that, you need to create a `Message` model and respective database table. You also define the relationship between the `Message` and `User` models.

Next you need a `Message` model, and the database table migration will be created along with it.

```
//code 11.3
$ php artisan make:model Message -m
```

Here is the model `Message`:

```
//code 11.4
//app/Message.php
<?php
```

```
namespace App;

use Illuminate\Database\Eloquent\Model;

class Message extends Model
{
    /**
     * Fields that are mass assignable
     *
     * @var array
     */
    protected $fillable = ['message', 'user_id'];

    /**
     * A message belong to a user
     *
     * @return \Illuminate\Database\Eloquent\Relations\BelongsTo
     */
    public function user()
    {
        return $this->belongsTo(User::class);
    }
}
```

One message belongs to a single user, and you have defined that in your `Message` model.

At the same time, you should define an inverse relationship in your `User` model too, as shown here:

```
//code 11.5
//app/User.php
<?php

namespace App;

use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
```

```
{
    use Notifiable;

    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = [
        'name', 'email', 'password',
    ];

    /**
     * The attributes that should be hidden for arrays.
     *
     * @var array
     */
    protected $hidden = [
        'password', 'remember_token',
    ];

    /**
     * A user can have many messages
     *
     * @return \Illuminate\Database\Eloquent\Relations\HasMany
     */
    public function messages()
    {
        return $this->hasMany(Message::class);
    }
}
```

One user can have many messages.

Now you are all are set, and you can work on the view templates where you display your messages and show the create form.

To do that, you need three view pages for showing and creating messages. I am not going to show all the parts of the view pages here for brevity. I will point out only the relevant parts.

```
//code 11.6
//resources/views/messages/index.blade.php
<div class="blog-post">
            <ul class="list-group">
                 @foreach ($messages as $message)
                 <li class="list-group-item"><h2 class="blog-post-title">
                         <a href="/messages/{{ $message->id }}">{{
                         $message->message }}</a>
                     </h2>
                 </li>
                 <li><a href="/messages/{{ $message->id }}/edit">Edit</a>
                 </li>
                 @endforeach
            </ul>
                     </div>
```

Next you need to see the separate message in your show template, as shown here:

```
//code 11.7
//resources/views/messages/show.blade.php
<div class="col-md-8 blog-main col-lg-8 blog-main col-sm-8 blog-main">
          <h3 class="pb-3 mb-4 font-italic border-bottom">
              <a href="/messages">The Message</a>
          </h3>
          <h3 class="pb-3 mb-4 font-italic border-bottom">
             {{ $message->message }} posted by {{ $message->user['name'] }}
          </h3>
        </div>
```

Finally, you want to create messages using a form, as shown here:

```
//code 11.8
//resources/views/messages/create.blade.php
<div class="col-md-8 blog-main col-lg-8 blog-main col-sm-8 blog-main">
          <h3 class="pb-3 mb-4 font-italic border-bottom">
            All Messages
          </h3>
            <div class="blog-post">
            <h2 class="blog-post-title"></h2>
             <form method="post" action="{{ route('messages.store') }}">
             {{ csrf_field() }}
             <div class="form-group">
             <label for="message">Name<span class="required">*</span>
             </label>
             <input   placeholder="Enter message"
             id="message"
             required
             name="message"
             spellcheck="false"
             class="form-control"
             />
             </div>

             <div class="form-group">
             <input type="submit" class="btn btn-primary"
             value="Submit"/>
             </div>
             </form>
             </div>
        </div>
```

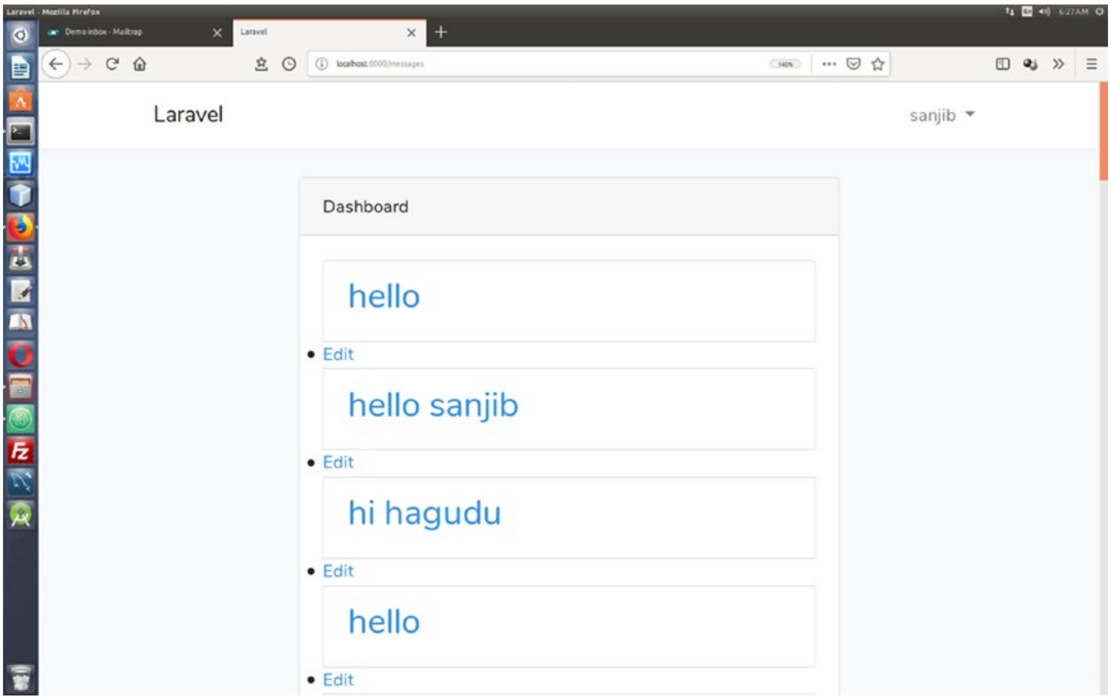
This is a simple form to create a single message. Each user has to log in, and only then can the user post a message through the controller.

You need to register your routes, which is quite simple.

```
//code 11.9
//routes/web.php
Route::get('/', function () {
    return view('welcome');
});

Auth::routes();

Route::resource('messages', 'CreatemessageController');

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

Here you are concerned only about the route `messages` and the resourceful controller `CreatemessageController`. When you type `http://localhost:8000/messages`, it takes you to the login page. Any user can log into the system and create messages. You get to that form at the URL `http://localhost:8000/messages/create`.

You have a few messages to display first (see Figure 11-1).



hello
- Edit

hello sanjib
- Edit

hi hagudu
- Edit

hello
- Edit

***Figure 11-1.** There are several messages that one user called sanjib has created*

The user creates messages using the form page shown in Figure 11-2.

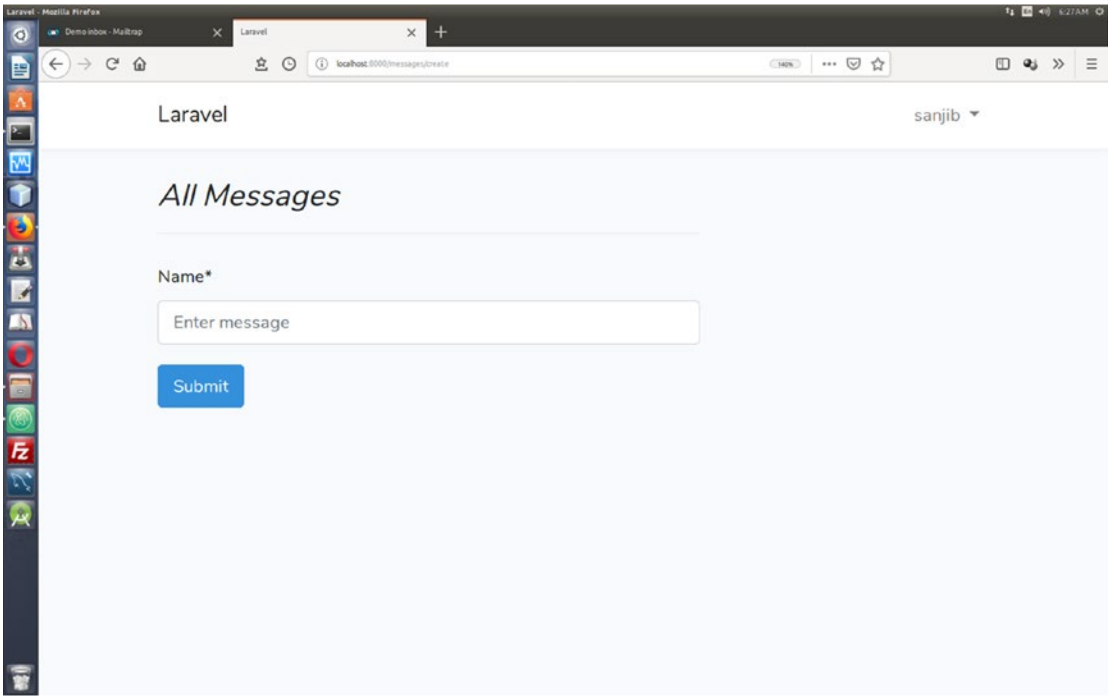


Figure 11-2. *The user can create messages here*

While a user creates a message, an event automatically fires through the controller `store` method. Let's create the event manually first; then you will see how events can be generated automatically. To understand events, listeners, and subscribers, you need to install a fresh Laravel application, called `laraeventandlisteners`, in your `code` directory. The source code is available in the code download; please take a look while studying this chapter.

```
//code 11.10
ss@ss-H81M-S1:~/code/laraeventandlisteners$ php artisan make:event
AboutTheUser
Event created successfully.
```

The event AboutTheUser has been created in the app/Events directory. Let's see the code first; it's as follows:

```
//code 11.11
//app/Events/AboutTheUser.php
<?php

namespace App\Events;

use Illuminate\Broadcasting\Channel;
use Illuminate\Queue\SerializesModels;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Foundation\Events\Dispatchable;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
use Illuminate\Support\Facades\Auth;
use App\User;

class AboutTheUser
{
    use Dispatchable, InteractsWithSockets, SerializesModels;

public $name;

    /**
     * Create a new event instance.
     *
     * @return void
     */
    public function __construct($name)
    {
      $user = Auth::user();
      $this->name = $name;
    }

    /**
     * Get the channels the event should broadcast on.
     *
     * @return \Illuminate\Broadcasting\Channel|array
     */
    public function broadcastOn()
    {
        return new PrivateChannel('channel-name');
    }
}
```

In the event constructor, you pass an authorized user object so that this event fires only for the authorized users.

You need to create several listeners for this event, as shown here:

```
//code 11.12
ss@ss-H81M-S1:~/code/laraeventandlisteners$ php artisan make:listener
SendMailForNewMessage --event=AboutTheUser
Listener created successfully.
ss@ss-H81M-S1:~/code/laraeventandlisteners$ php artisan make:listener
AlertForNewMessage
Listener created successfully.
ss@ss-H81M-S1:~/code/laraeventandlisteners$ php artisan make:listener
AboutTheUser
Listener created successfully.
```

You have created three listeners for this event. Now you need to register those event listeners to the `EventServiceProvider` so that the `listen` property, inside it, can contain an array of all events (keys) and their listeners (values).

You can add as many events and respective listeners to this array as you need.

Let's see the original `EventServiceProvider` code that comes with Laravel. You are going to change it soon to fit your newly created events and listeners.

```
//code 11.13
//app/Providers/EventServiceProvider.php
//original one

<?php

namespace App\Providers;
```

```
use Illuminate\Support\Facades\Event;
use Illuminate\Auth\Events\Registered;
use Illuminate\Auth\Listeners\SendEmailVerificationNotification;
use Illuminate\Foundation\Support\Providers\EventServiceProvider as
ServiceProvider;

class EventServiceProvider extends ServiceProvider
{
    /**
     * The event listener mappings for the application.
     *
     * @var array
     */
    protected $listen = [
        Registered::class => [
            SendEmailVerificationNotification::class,
        ],
    ];

    /**
     * Register any events for your application.
     *
     * @return void
     */
    public function boot()
    {
        parent::boot();

        //
    }
}
```

Next, you will register your event listeners here, and after the change it looks like this:

```
//code 11.14
//app/Providers/EventServiceProvider.php
//changed code where the listen property takes new values

<?php
```

```
namespace App\Providers;

use Illuminate\Support\Facades\Event;
use Illuminate\Auth\Events\Registered;
use App\Events\AboutTheUser;
use App\Listeners\SendMailForNewMessage;
use App\Listeners\AlertForNewMessage;
use App\Listeners\AboutTheUser;
use Illuminate\Auth\Listeners\SendEmailVerificationNotification;
use Illuminate\Foundation\Support\Providers\EventServiceProvider as
ServiceProvider;

class EventServiceProvider extends ServiceProvider
{
    /**
     * The event listener mappings for the application.
     *
     * @var array
     */
    protected $listen = [
        AboutTheUser::class => [
            AlertForNewMessage::class,
            AboutTheUser::class,
            SendMailForNewMessage::class,

        ],
    ];

    /**
     * Register any events for your application.
     *
     * @return void
     */
    public function boot()
    {
        parent::boot();
        //
    }
}
```

The entire `listen` property has been changed, and the event key `AboutTheUser::class` points to three listener values.

- `AlertForNewMessage::class`
- `AboutTheUser::class`
- `SendMailForNewMessage::class`

Don't forget to use the proper namespace at the top of the event service provider.

```
use App\Events\AboutTheUser;
use App\Listeners\SendMailForNewMessage;
use App\Listeners\AlertForNewMessage;
```

Now, you are ready to add code to the respective listeners. Let's try to understand one key concept of Laravel here. You want to create an application based on the SOLID design principle. You could have done the same thing through the `store` method of your controller. But in that case, the code meant for the listeners would have added to the controller `store` method, making it tightly coupled.

You do not want to do that for one single reason: each class should have single responsibility. In other words, each class should have single task to accomplish.

In the next section, you will see how the event fires and the tasks are delegated to the respective listeners. You will also see the listeners' code.

Receiving Messages

Now you have three listeners in the `app/Listeners` directory.

```
//code 11.15
//app/Listeners/AboutTheUser.php
<?php

namespace App\Listeners;

use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;
```

```
class AboutTheUser
{

    /**
     * Handle the event.
     *
     * @param  object  $event
     * @return void
     */
    public function handle($event)
    {
        dump('Check whether the user in your friend lists');
    }
}
```

Just to make the concept clear, I have dumped the data. You can use a JavaScript framework like Vue here so that each logged-in user can get notified and know about the user who has posted the new post. The next listener will alert the users about the message.

```
//code 11.16
//app/Listeners/AlertForNewMessage.php
<?php

namespace App\Listeners;

use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Support\Facades\Mail;
use Illuminate\Support\Facades\Auth;
use App\User;

class AlertForNewMessage
{

    /**
     * Handle the event.
     *
     * @param  object  $event
     * @return void
     */
    public function handle($event)
    {
        //the event will handle the mail event
        dump('A new mesage has been posted on your post');
    }
}
```

I have dumped the message here like the previous one, although in the next listener, `SendMailForNewMessage`, I will show how to test a real mail that will be sent when a user posts a new message.

Let's see the `SendMailForNewMessage` code first.

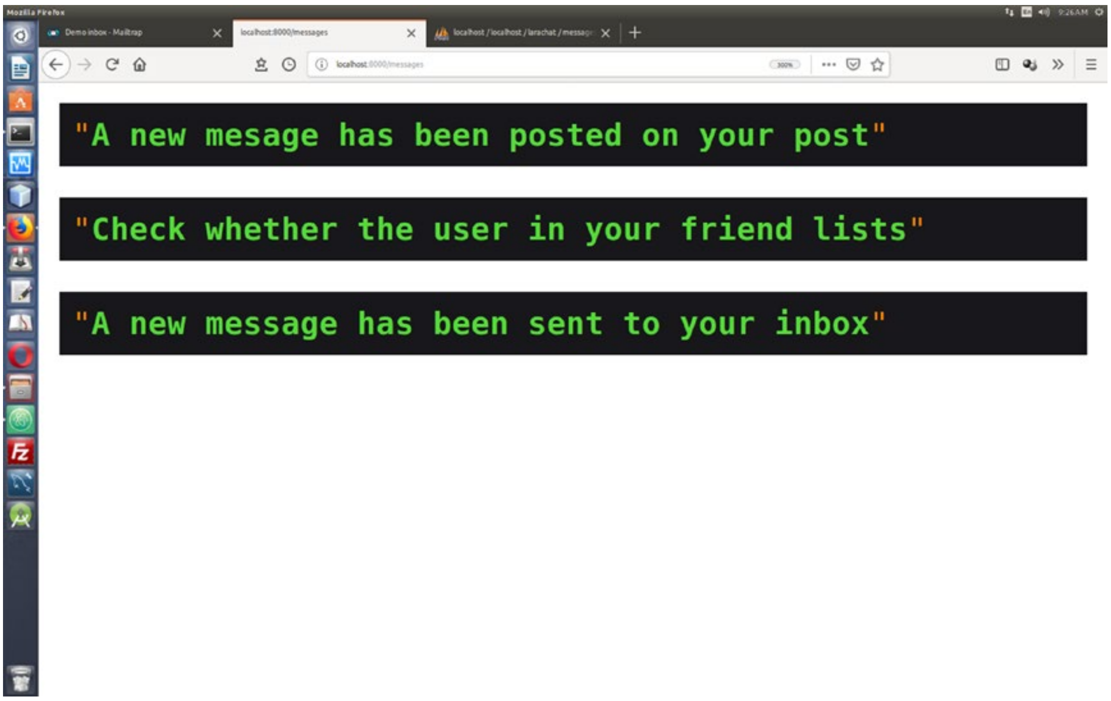
```
//code 11.17
//app/Listeners/SendMailForNewMessage.php
<?php

namespace App\Listeners;

use App\Events\NewMessagePosted;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Support\Facades\Mail;
use Illuminate\Support\Facades\Auth;
use App\Mail\MailForNewMessage;

class SendMailForNewMessage
{
    /**
     * Handle the event.
     *
     * @param  NewMessagePosted  $event
     * @return void
     */
    public function handle(NewMessagePosted $event)
    {
        $user = Auth::user();
        Mail::to($user->email)->send(new MailForNewMessage());
    }
}
```

To test a real mail, you need to set up your environment through Mailtrap (you did this in the previous chapter). In the `.env` file, I have included my Mailtrap key and other requirements that are necessary for sending a mail.

```
//code 11.18
//.env
MAIL_DRIVER=smtp
MAIL_HOST=smtp.mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=*****************
MAIL_PASSWORD=*****************
MAIL_ENCRYPTION=tls
```

Creating a Mailtrap account is easy, and it is free to use their sandbox. So, please go ahead and do that and change the username and password fields.

If I had not used a real mail environment using Mailtrap and instead just dumped the data in the `SendMailForNewMessage` listener, you would see the screen shown in Figure 11-3.



```
"A new mesage has been posted on your post"

"Check whether the user in your friend lists"

"A new message has been sent to your inbox"
```

Figure 11-3. *Dumping all data when the event fires*

Instead, you can use an `artisan` command to create a mail class called `MailForNewMessage`.

```
//code 11.19
ss@ss-H81M-S1:~/code/laraeventandlisteners$ php artisan make:mail
MailForNewMessage --markdown emails.new-message
Mail created successfully.
```

You use the `markdown` flag to create a `new-message` blade template in the `resources/views/emails` folder.

Let's see the newly created `MailForNewMessage` class first, as shown here:

```
//code 11.20
//app/Mail/ MailForNewMessage.php
<?php

namespace App\Mail;
```

```
use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;
use Illuminate\Contracts\Queue\ShouldQueue;

class MailForNewMessage extends Mailable
{
    use Queueable, SerializesModels;

    /**
     * Build the message.
     *
     * @return $this
     */
    public function build()
    {
        return $this->markdown('emails.new-message');
    }
}
```

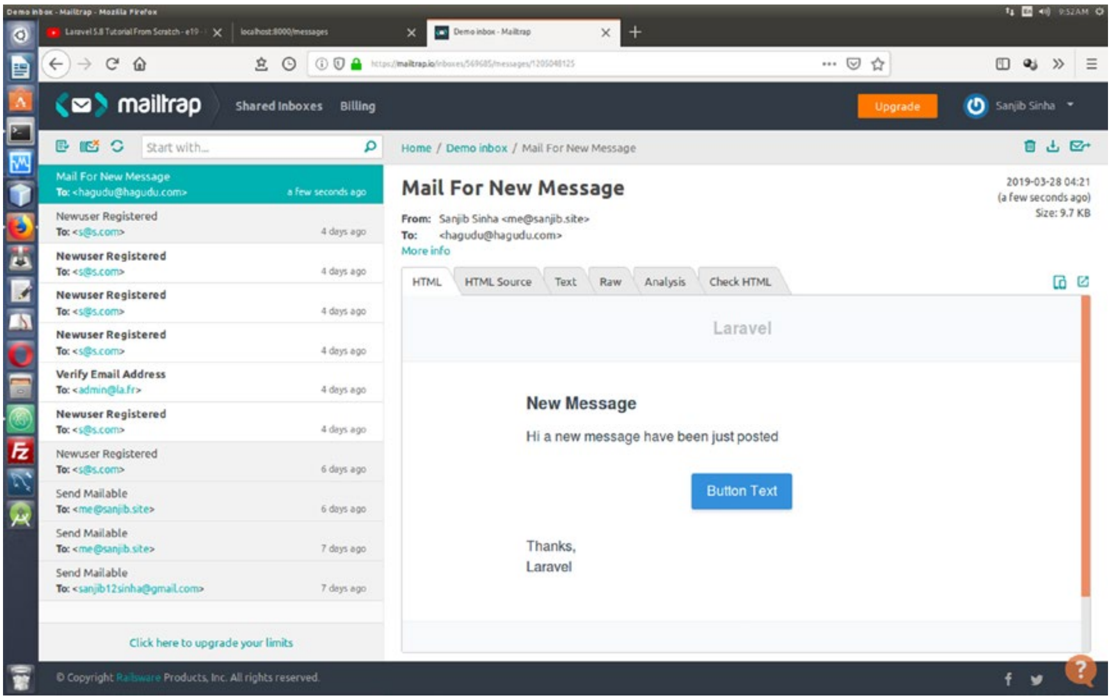
In the `resources/views/emails/new-message.blade.php` file, you have some autogenerated code that looks like this:

```
//code 11.21
@component('mail::message')
# New Message

Hi a new message have been just posted

@component('mail::button', ['url' => ''])
Button Text
@endcomponent

Thanks,<br>
{{ config('app.name') }}
@endcomponent
```

You are free to edit the code, and you can even completely change the whole HTML look of this page. I have kept it as is just to wrap it up quickly.

Now what happens when a user creates or posts a message?

An e-mail has been sent, and you have a message in your Mailtrap inbox (Figure 11-4).



Mail For New Message
To: <hagudu@hagudu.com> a few seconds ago
Newuser Registered
To: <s@s.com> 4 days ago
Newuser Registered
To: <s@s.com> 4 days ago
Newuser Registered
To: <s@s.com> 4 days ago
Newuser Registered
To: <s@s.com> 4 days ago
Verify Email Address
To: <admin@la.fr> 4 days ago
Newuser Registered
To: <s@s.com> 4 days ago
Newuser Registered
To: <s@s.com> 6 days ago
Send Mailable
To: <me@sanjib.site> 6 days ago
Send Mailable
To: <me@sanjib.site> 7 days ago
Send Mailable
To: <sanjib12sinha@gmail.com> 7 days ago

Mail For New Message

2019-03-28 04:21
(a few seconds ago)
Size: 9.7 KB

From: Sanjib Sinha <me@sanjib.site>
To: <hagudu@hagudu.com>
More info

Laravel

New Message

Hi a new message have been just posted

Button Text

Thanks,
Laravel

© Copyright Railsware Products, Inc. All rights reserved.

Figure 11-4. *The mail has been sent, and the new-message.blade.php page is reflected in your Mailtrap sandbox*

So, you have successfully fired the event, and the event has broadcast the tasks to the listeners. After that, the listeners take actions accordingly.

In conclusion, you could say events and listeners are an integral part of your Laravel application for many reasons. However, the most important part is that they make your application loosely coupled. That reflects the SOLID design principle, which is the goal.

Now you can add a `$subscriber` property to your `EventServiceProvider` class as you have added the `$listen` property just like this:

```
protected $listen = [
        Registered::class => [
            SendEmailVerificationNotification::class,
        ],
    ];
```

The `$subscriber` property will look like this:

```
/**
     * The subscriber classes to register.
     *
     * @var array
     */
    protected $subscribe = [
        'App\Listeners\EventServiceProvider ',
    ];
```

After that, you can create a `subscriber` class in your `app/Listeners/NewSubscriber.php` directory and add a `subscribe` method where you can register multiple events.

```
public function subscribe($events)
    {
        $events->listen(
            //code
        );

        $events->listen(
            //code
        );
        $events->listen(
            //code
        );

        $events->listen(
            //code
        );

    }
```

You can add multiple events this way.

One thing should be clear by now. You can call or use an event for multiple purposes. In some cases, when your queued job fails, you can use the `Queue::failing` method. This event is a great opportunity to notify your team via e-mail or any chatting app.

`AppServiceProvider` is provided with the Laravel, where you can attach a callback to any event that may fail.

Consider this code:

```
//app/Providers/AppServiceProvider.php
<?php

namespace App\Providers;

use Queue;
use Illuminate\Queue\Events\JobFailed;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Queue::failing(function (JobFailed $event) {
            // here goes your all events name and code
        });
    }

    /**
     * Register the service provider.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

To summarize, event subscribers are classes that can subscribe to multiple events from within the class itself, and this gives you a chance to define several event handlers within that class. At the same time, there are several options to use events to handle the failed jobs and notify the users at the same time.

Autogenerating Events

Before concluding this chapter, I want to show you how to autogenerate the events. You need to manually add the event's key and listener values to the `listen` property in the event service provider first. After that, you just run this command:

```
//code 11.22
$ php artisan event:generate
```

This will create all the missing events and listeners based on registration in your event service provider.

CHAPTER 12

Working with APIs

What allows a third party to easily interact with a Laravel application's data? The answer is an API.

Usually the API is based on JSON and REST or is REST-like. You can easily work with JSON in your Laravel application, which gives Laravel a big advantage over other PHP frameworks. Without an API you cannot interact with any third-party software that is written in a different language and that works on different platforms. So, writing APIs is a common task that Laravel developers do in their jobs.

Another advantage of Laravel is that its resource controllers are already structured around REST verbs and patterns. This makes Laravel developers' lives much easier.

In this chapter, you will learn to write an API.

What Is REST?

Representational State Transfer (REST) is an architectural style for building APIs.

There are some heated arguments over the definition of REST in the computer world. Please do not get overwhelmed by the definition or get caught in the crossfire. With Laravel, when I talk about REST-like APIs, I generally mean they have a few common characteristics; for example, they are structured around resources, and the APIs can be represented by simple URIs.

The URI `http://localhost:8000/articles` is a representation of all articles. The URI `http:://localhost:8000/article/2` represents the second article. The representation of second article goes on just like normal URI representation.

The stateless condition of APIs makes a big difference. Between requests, there is no persistent session authentication, which means that each request needs to authenticate itself. Finally, the major advantage is it can return JSON, which the server understands. That is the reason why a third party can easily interact with a Laravel application.

S. Sinha, *Beginning Laravel*, https://doi.org/10.1007/978-1-4842-4991-8_12

The main purpose of building an API is to enable another application to communicate with your application without any issues. The server knows XML and JSON; however, JSON is the most popular choice now. So, when you return your data in JSON, the ease of communication increases.

Creating an API

To create an API, you need controllers and models as usual. But, at the same time, you need to transform your models and model collections into JSON. For that, you want Laravel's resource classes. They allow you to transform data into JSON.

You can imagine your resource classes as a transformation layer that sits between your Eloquent models and the JSON responses.

Let's start from scratch. The first step is to create a fresh Laravel project.

```
//code 12.1
$ composer create-project --prefer-dist laravel/laravel apilara
```

Let's say you want to create an `Article` API. So, you need a model, database table, and controller.

```
//code 12.2
//creating a controller
$ php artisan make:controller ArticleController --resource
Controller created successfully.
```

Next, you need an `Article` model and a database table, as shown here:

```
//code 12.3
$ php artisan make:model Article -m
```

The database table should have two fields, `title` and `body`, as shown here:

```
//code 12.4
<?php

use Illuminate\Support\Facades\Schema;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;
```

```
class CreateArticlesTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('articles', function (Blueprint $table) {
            $table->bigIncrements('id');
            $table->string('title');
            $table->text('body');
            $table->timestamps();
        });
    }

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

Before building an API, you should fill the `Article` table with some fake data with the help of a database seeder. To do this, you need to create `ArticlesTableSeeder` and `ArticleFactory`, as shown here:

```
//code 12.5
$ php artisan make:seeder ArticlesTableSeeder
Seeder created successfully.
//code 12.6
$ php artisan make:factory ArticleFactory
Factory created successfully
```

Let's change the code of both ArticlesTableSeeder and ArticleFactory, as shown here:

```
//code 12.7
//database/seeds/ ArticlesTableSeeder.php
<?php

use Illuminate\Database\Seeder;
use App\Article;

class ArticlesTableSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        factory(Article::class, 30)->create();
    }
}
```

This will create 30 articles. Next, you need to prepare your factory class to fill up the articles table, as shown here:

```
//code 12.8
// database/factories/ArticleFactory.php
<?php

use Faker\Generator as Faker;

$factory->define(App\Article::class, function (Faker $faker) {
    return [
        'title' => $faker->text(50),
        'body' => $faker->text(300)
    ];
});
```

The title property will have a maximum of 50 characters, and for the body you will be content with 300 characters for demonstration purposes.

Let's fill the database table, as shown here:

```
//code 12.9
$ php artisan db:seed
Seeding: ArticlesTableSeeder
Database seeding completed successfully.
```

After the completion of database seeding, you will generate the resource class. To do that, you will use the make:resource Artisan command.

```
//code 12.10
$ php artisan make:resource Article
Resource created successfully.
```

These resources are created to transform the individual models. You may want to generate resources that are responsible for transforming collections of models. This allows you to include links and other meta information in your response.

To create a resource collection, either you can use the --collection flag or you can include the word Collection in the resource name. The word Collection will indicate to Laravel that it should create a collection resource. Collection resources extend the Illuminate\Http\Resources\Json\ResourceCollection class. The relevant artisan command will look like this:

```
//code 12.13
$ php artisan make:resource Article --collection
$ php artisan make:resource ArticleCollection
```

Later in this chapter I will discuss collections and show how they work.

So, you have created a resource called Article. You can find it in the app/Http/Resources directory of your application. Resources extend the Illuminate\Http\Resources\Json\JsonResource class.

```
//code 12.14
//app/Http/Resources/Article.php
<?php

namespace App\Http\Resources;
```

```
use Illuminate\Http\Resources\Json\JsonResource;

class Article extends JsonResource
{
    /**
     * Transform the resource into an array.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return array
     */
    public function toArray($request)
    {
             return parent::toArray($request);
    }
}
```

As you can see in the previous code, every resource class defines a `toArray` method, which returns the array of attributes that should be converted to JSON when sending the response. Laravel takes care of that in the background.

You are not ready yet. You need to register your API routes in the `routes/api.php` file, like this:

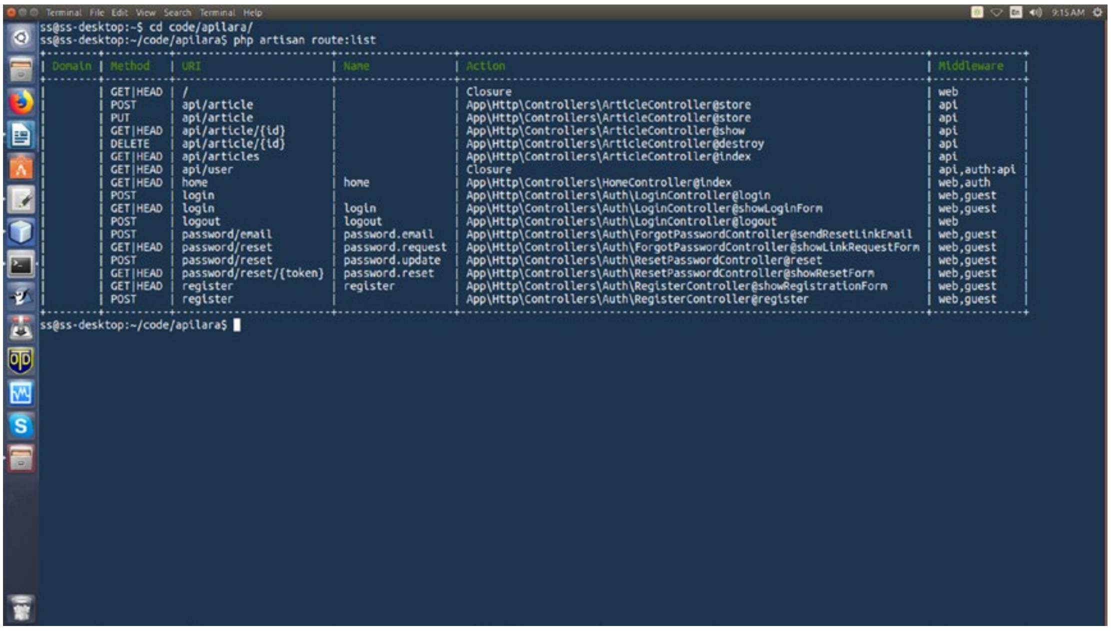
```
//code 12.15
//routes/api.php
<?php

use Illuminate\Http\Request;


/*
|--------------------------------------------------------------------------
| API Routes
|--------------------------------------------------------------------------
|
| Here is where you can register API routes for your application. These
| routes are loaded by the RouteServiceProvider within a group which
| is assigned the "api" middleware group. Enjoy building your API!
|
*/
```

```
Route::middleware('auth:api')->get('/user', function (Request $request) {
    return $request->user();
});

//list articles
Route::get('articles', 'ArticleController@index');
//list single article
Route::get('article/{id}', 'ArticleController@show');
//create new article
Route::post('article', 'ArticleController@store');
//update articles
Route::put('article', 'ArticleController@store');
//delete articles
Route::delete('article/{id}', 'ArticleController@destroy');
```

You see the URI, method, and action parts of your routes. If you want a full list, you can issue this `artisan` command in your terminal:

```
$ php artisan route:list
```

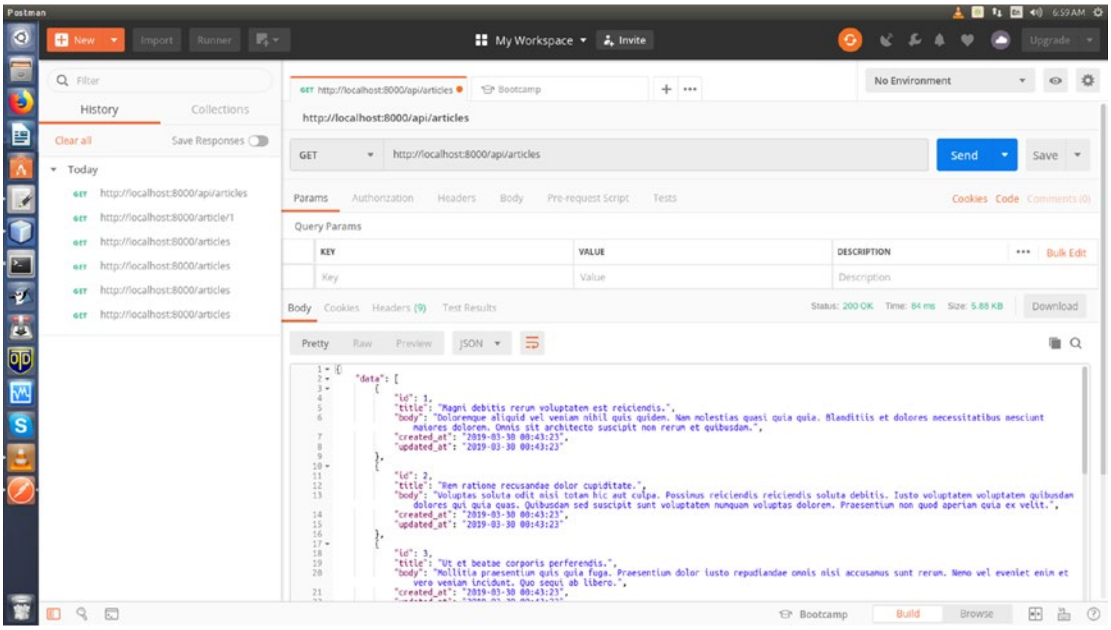
Now you can see the full route list (Figure 12-1).



```
ss@ss-desktop:~$ cd code/apilara/
ss@ss-desktop:~/code/apilara$ php artisan route:list
+--------+----------+------------------------+------------------+------------------------------------------------------------------------+--------------+
| Domain | Method   | URI                    | Name             | Action                                                                 | Middleware   |
+--------+----------+------------------------+------------------+------------------------------------------------------------------------+--------------+
|        | GET|HEAD | /                      |                  | Closure                                                                | web          |
|        | POST     | api/article            |                  | App\Http\Controllers\ArticleController@store                           | api          |
|        | PUT      | api/article            |                  | App\Http\Controllers\ArticleController@store                           | api          |
|        | GET|HEAD | api/article/{id}       |                  | App\Http\Controllers\ArticleController@show                            | api          |
|        | DELETE   | api/article/{id}       |                  | App\Http\Controllers\ArticleController@destroy                         | api          |
|        | GET|HEAD | api/articles           |                  | App\Http\Controllers\ArticleController@index                           | api          |
|        | GET|HEAD | api/user               |                  | Closure                                                                | api,auth:api |
|        | GET|HEAD | home                   | home             | App\Http\Controllers\HomeController@index                              | web,auth     |
|        | POST     | login                  |                  | App\Http\Controllers\Auth\LoginController@login                        | web,guest    |
|        | GET|HEAD | login                  | login            | App\Http\Controllers\Auth\LoginController@showLoginForm                | web,guest    |
|        | POST     | logout                 | logout           | App\Http\Controllers\Auth\LoginController@logout                       | web          |
|        | POST     | password/email         | password.email   | App\Http\Controllers\Auth\ForgotPasswordController@sendResetLinkEmail  | web,guest    |
|        | GET|HEAD | password/reset         | password.request | App\Http\Controllers\Auth\ForgotPasswordController@showLinkRequestForm | web,guest    |
|        | POST     | password/reset         | password.update  | App\Http\Controllers\Auth\ResetPasswordController@reset                | web,guest    |
|        | GET|HEAD | password/reset/{token} | password.reset   | App\Http\Controllers\Auth\ResetPasswordController@showResetForm        | web,guest    |
|        | GET|HEAD | register               | register         | App\Http\Controllers\Auth\RegisterController@showRegistrationForm      | web,guest    |
|        | POST     | register               |                  | App\Http\Controllers\Auth\RegisterController@register                  | web,guest    |
+--------+----------+------------------------+------------------+------------------------------------------------------------------------+--------------+
ss@ss-desktop:~/code/apilara$
```

Figure 12-1. *The API route list in your terminal*

After starting the local server, you can open Postman to get the response (Figure 12-2). Postman is a tool that can send requests to an API and show you the response. In other words, it is a fancy version of cURL.



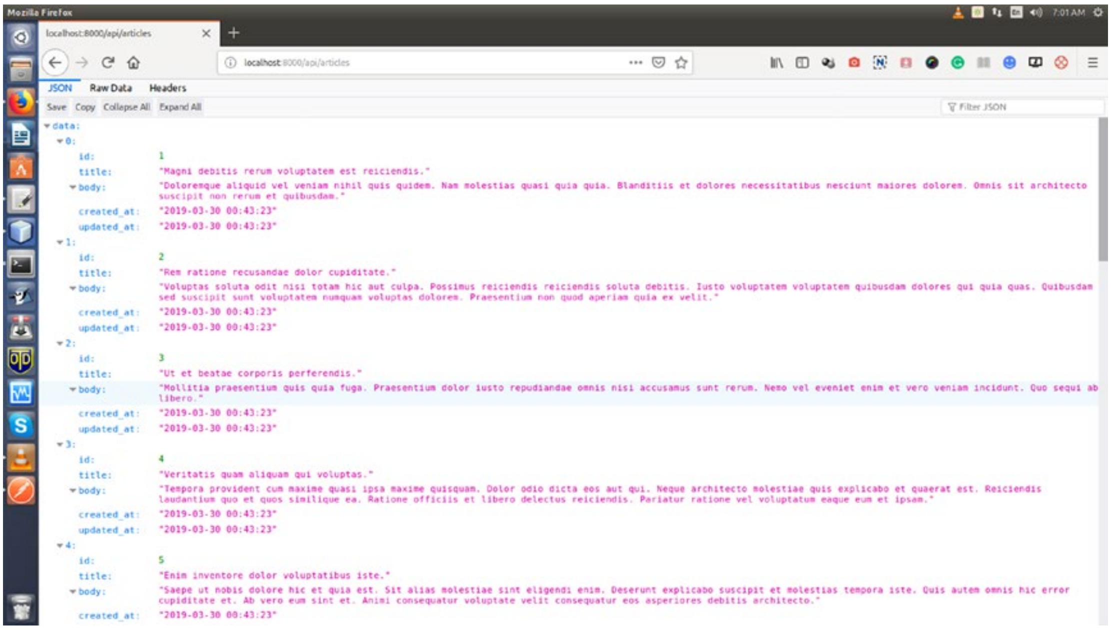
GET http://localhost:8000/api/articles

Status: 200 OK Time: 84 ms Size: 5.88 KB

```
{
    "data": [
        {
            "id": 1,
            "title": "Magni debitis rerum voluptatem est reiciendis.",
            "body": "Dolorenque aliquid vel veniam nihil quis quidem. Nam molestias quasi quia quia. Blanditiis et dolores necessitatibus nesciunt
                maiores dolorem. Omnis sit architecto suscipit non rerum et quibusdam.",
            "created_at": "2019-03-30 00:43:23",
            "updated_at": "2019-03-30 00:43:23"
        },
        {
            "id": 2,
            "title": "Rem ratione recusandae dolor cupiditate.",
            "body": "Voluptas soluta odit nisi totam hic aut culpa. Possimus reiciendis reiciendis soluta debitis. Iusto voluptatem voluptatem quibusdam
                dolores qui quia quas. Quibusdam sed suscipit sunt voluptatem numquam voluptas dolorem. Praesentium non quod aperiam quia ex velit.",
            "created_at": "2019-03-30 00:43:23",
            "updated_at": "2019-03-30 00:43:23"
        },
        {
            "id": 3,
            "title": "Ut et beatae corporis perferendis.",
            "body": "Mollitia praesentium quis quia fuga. Praesentium dolor iusto repudiandae omnis nisi accusamus sunt rerum. Nemo vel eveniet enim et
                vero veniam incidunt. Quo sequi ab libero.",
            "created_at": "2019-03-30 00:43:23",
```

Figure 12-2. *The JSON response of the Article API in Postman*

You can get the same output in your regular web browser also. The URL is `http://localhost:8000/api/articles`. See Figure 12-3.



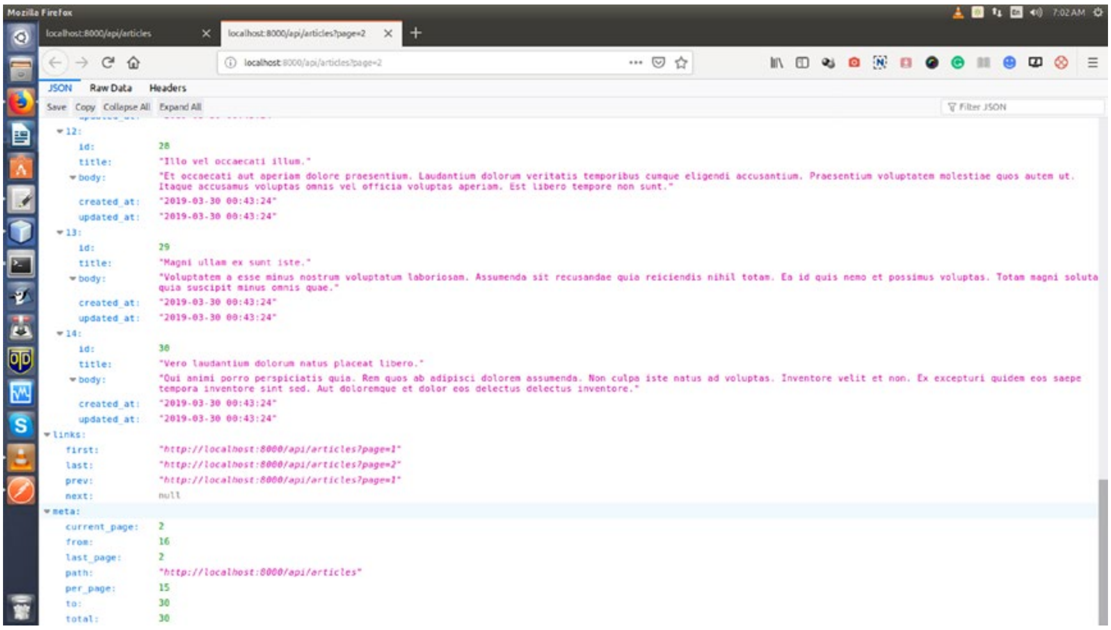
```
data:
  0:
    id:           1
    title:        "Magni debitis rerum voluptatem est reiciendis."
    body:         "Doloremque aliquid vel veniam nihil quis quidem. Nam molestias quasi quia quia. Blanditiis et dolores necessitatibus nesciunt maiores dolorem. Omnis sit architecto
                  suscipit non rerum et quibusdam."
    created_at:   "2019-03-30 00:43:23"
    updated_at:   "2019-03-30 00:43:23"
  1:
    id:           2
    title:        "Rem ratione recusandae dolor cupiditate."
    body:         "Voluptas soluta odit nisi totam hic aut culpa. Possimus reiciendis reiciendis soluta debitis. Iusto voluptatem voluptatem quibusdam dolores qui quia quas. Quibusdam
                  sed suscipit sunt voluptatem numquam voluptas dolorem. Praesentium non quod aperiam quia ex velit."
    created_at:   "2019-03-30 00:43:23"
    updated_at:   "2019-03-30 00:43:23"
  2:
    id:           3
    title:        "Ut et beatae corporis perferendis."
    body:         "Mollitia praesentium quis quia fuga. Praesentium dolor iusto repudiandae omnis nisi accusamus sunt rerum. Nemo vel eveniet enim et vero veniam incidunt. Quo sequi ab
                  libero."
    created_at:   "2019-03-30 00:43:23"
    updated_at:   "2019-03-30 00:43:23"
  3:
    id:           4
    title:        "Veritatis quam aliquam qui voluptas."
    body:         "Tempora provident cum maxime quasi ipsa maxime quisquam. Dolor odio dicta eos aut qui. Neque architecto molestiae quis explicabo et quaerat est. Reiciendis
                  laudantium quo et quos similique ea. Ratione officiis et libero delectus reiciendis. Pariatur ratione vel voluptatum eaque eum et ipsam."
    created_at:   "2019-03-30 00:43:23"
    updated_at:   "2019-03-30 00:43:23"
  4:
    id:           5
    title:        "Enim inventore dolor voluptatibus iste."
    body:         "Saepe ut nobis dolore hic et quia est. Sit alias molestiae sint eligendi enim. Deserunt explicabo suscipit et molestias tempora iste. Quis autem omnis hic error
                  cupiditate et. Ab vero eum sint et. Animi consequatur voluptate velit consequatur eos asperiores debitis architecto."
    created_at:   "2019-03-30 00:43:23"
```

Figure 12-3. *The same JSON response in reply to a GET request in a normal browser*

Let's take a look at the ArticleController index method, as shown here:

```
//code 12.16
//app/Http/Controllers/ArticleController.php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use App\Http\Requests;
use App\Article;
use App\Http\Resources\Article as ArticleResource;

class ArticleController extends Controller
{
    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        //get articles
        $articles = Article::paginate(15);

        //return collection of articles as resource
        return ArticleResource::collection($articles);
    }

    /**
     * Display the specified resource.
     *
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function show($id)
    {
        //get article
        $article = Article::findOrFail($id);

        //returning single article as a resource
        return new ArticleResource($article);
    }
}
```

For demonstration purposes, I have shown only two methods here: `index` and `show`. Since I have used the `paginate()` method, each page shows you 15 articles. Figure 12-4 shows the second page.



localhost:8000/api/articles
localhost:8000/api/articles?page=2
localhost:8000/api/articles?page=2
JSON
Raw Data
Headers
Save
Copy
Collapse All
Expand All
Filter JSON

```
▾ 12:
    id:            28
    title:         "Illo vel occaecati illum."
  ▾ body:          "Et occaecati aut aperiam dolore praesentium. Laudantium dolorum veritatis temporibus cumque eligendi accusantium. Praesentium voluptatem molestiae quos autem ut.
                   Itaque accusamus voluptas omnis vel officia voluptas aperiam. Est libero tempore non sunt."
    created_at:    "2019-03-30 00:43:24"
    updated_at:    "2019-03-30 00:43:24"
▾ 13:
    id:            29
    title:         "Magni ullam ex sunt iste."
  ▾ body:          "Voluptatem a esse minus nostrum voluptatum laboriosam. Assumenda sit recusandae quia reiciendis nihil totam. Ea id quis nemo et possimus voluptas. Totam magni soluta
                   quia suscipit minus omnis quae."
    created_at:    "2019-03-30 00:43:24"
    updated_at:    "2019-03-30 00:43:24"
▾ 14:
    id:            30
    title:         "Vero laudantium dolorum natus placeat libero."
  ▾ body:          "Qui animi porro perspiciatis quia. Rem quos ab adipisci dolorem assumenda. Non culpa iste natus ad voluptas. Inventore velit et non. Ex excepturi quidem eos saepe
                   tempora inventore sint sed. Aut doloremque et dolor eos delectus delectus inventore."
    created_at:    "2019-03-30 00:43:24"
    updated_at:    "2019-03-30 00:43:24"
▾ links:
  first:           "http://localhost:8000/api/articles?page=1"
  last:            "http://localhost:8000/api/articles?page=2"
  prev:            "http://localhost:8000/api/articles?page=1"
  next:            null
▾ meta:
  current_page:    2
  from:            16
  last_page:       2
  path:            "http://localhost:8000/api/articles"
  per_page:        15
  to:              30
  total:           30
```

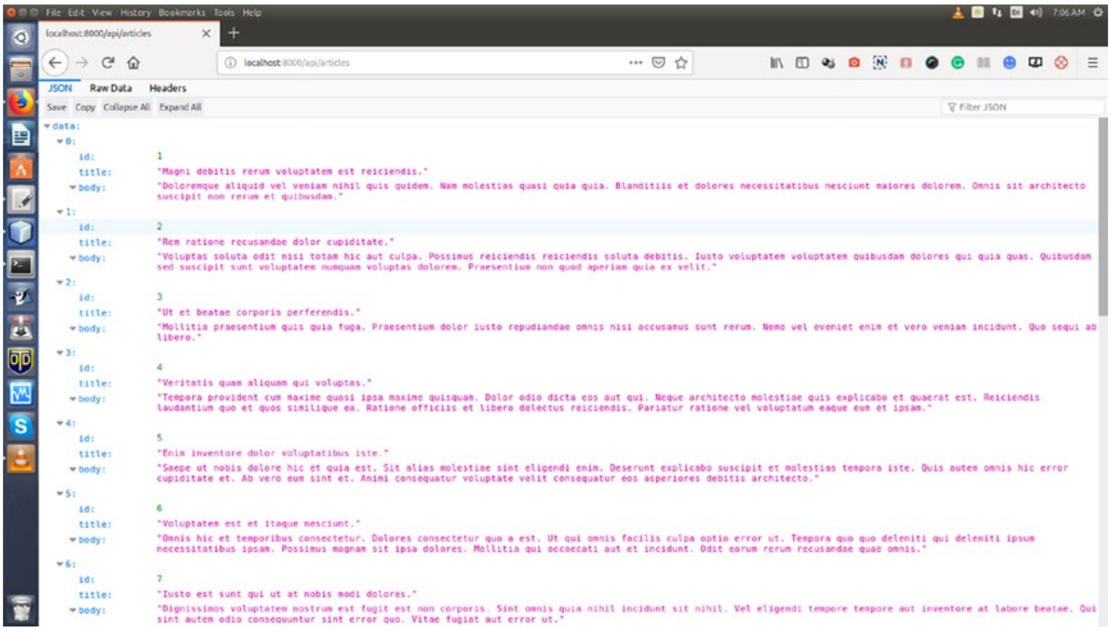
Figure 12-4. *The view of page 2 of the JSON response*

The URL is `http://localhost:8000/api/articles?page=2`.

Now you may not want to give your application user every output. You may want to omit `created_at` and `updated_at`. Laravel allows you to customize the JSON output. In that case, you can return the selected records like this:

```
//code 12.17
//app/Http/Resources/Article.php
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

class Article extends JsonResource
{
    /**
     * Transform the resource into an array.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return array
     */
    public function toArray($request)
    {
        return [
            'id' => $this->id,
            'title' => $this->title,
            'body' => $this->body
        ];
    }
}
```

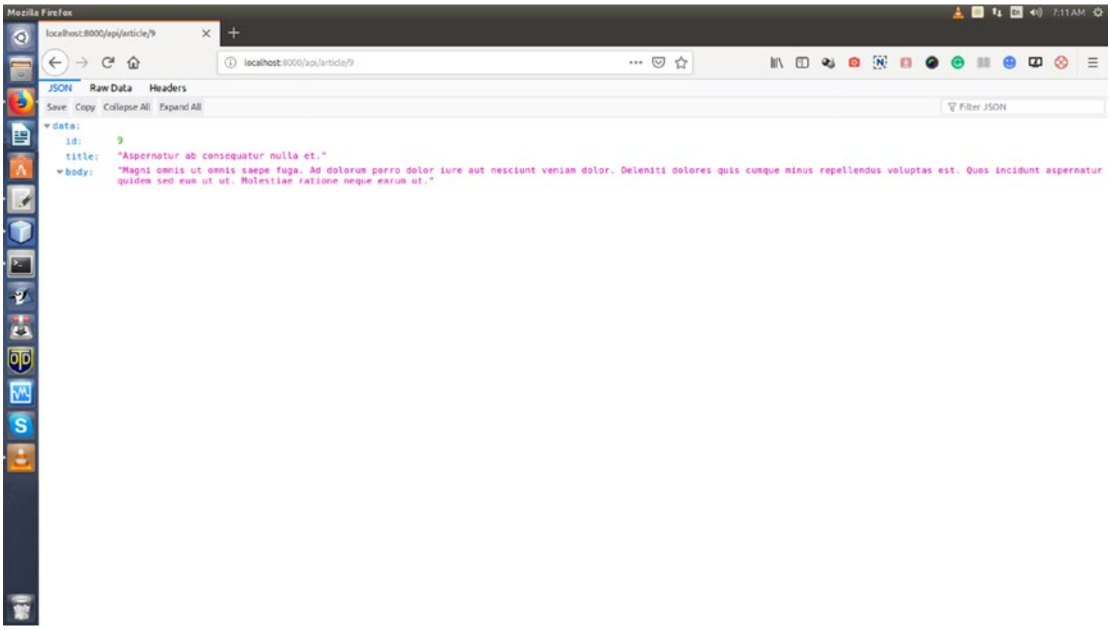
So, you are returning only `id`, `title`, and body and giving a customized look (Figure 12-5).


File Edit View History Bookmarks Tools Help
localhost:8000/api/articles
localhost:8000/api/articles
JSON Raw Data Headers
Save Copy Collapse All Expand All
Filter JSON

```
▾ data:
  ▾ 0:
      id:         1
      title:      "Magni debitis rerum voluptatem est reiciendis."
    ▾ body:       "Doloremque aliquid vel veniam nihil quis quidem. Nam molestias quasi quia quia. Blanditiis et dolores necessitatibus nesciunt maiores dolorem. Omnis sit architecto
                  suscipit non rerum et quibusdam."
  ▾ 1:
      id:         2
      title:      "Rem ratione recusandae dolor cupiditate."
    ▾ body:       "Voluptas soluta odit nisi totam hic aut culpa. Possimus reiciendis reiciendis soluta debitis. Iusto voluptatem voluptatem quibusdam dolores qui quia quas. Quibusdam
                  sed suscipit sunt voluptatem numquam voluptas dolorem. Praesentium non quod aperiam quia ex velit."
  ▾ 2:
      id:         3
      title:      "Ut et beatae corporis perferendis."
    ▾ body:       "Mollitia praesentium quis quia fuga. Praesentium dolor iusto repudiandae omnis nisi accusamus sunt rerum. Nemo vel eveniet enim et vero veniam incidunt. Quo sequi ab
                  libero."
  ▾ 3:
      id:         4
      title:      "Veritatis quam aliquam qui voluptas."
    ▾ body:       "Tempora provident cum maxime quasi ipsa maxime quisquam. Dolor odio dicta eos aut qui. Neque architecto molestiae quis explicabo et quaerat est. Reiciendis
                  laudantium quo et quos similique ea. Ratione officiis et libero delectus reiciendis. Pariatur ratione vel voluptatum eaque eum et ipsam."
  ▾ 4:
      id:         5
      title:      "Enim inventore dolor voluptatibus iste."
    ▾ body:       "Saepe ut nobis dolore hic et quia est. Sit alias molestiae sint eligendi enim. Deserunt explicabo suscipit et molestias tempora iste. Quis autem omnis hic error
                  cupiditate et. Ab vero eum sint et. Animi consequatur voluptate velit consequatur eos asperiores debitis architecto."
  ▾ 5:
      id:         6
      title:      "Voluptatem est et itaque nesciunt."
    ▾ body:       "Omnis hic et temporibus consectetur. Dolores consectetur quo a est. Ut qui omnis facilis culpa optio error ut. Tempora quo quo deleniti qui deleniti ipsum
                  necessitatibus ipsam. Possimus magnam sit ipsa dolores. Mollitia qui occaecati aut et incidunt. Odit earum rerum recusandae quae omnis."
  ▾ 6:
      id:         7
      title:      "Iusto est sunt qui ut at nobis modi dolores."
    ▾ body:       "Dignissimos voluptatem nostrum est fugit est non corporis. Sint omnis quia nihil incidunt sit nihil. Vel eligendi tempore tempore aut inventore at labore beatae. Qui
                  sint autem odio consequuntur sint error quo. Vitae fugiat aut error ut."
```

Figure 12-5. *The customized JSON response*

You can use the `show` method to have one article display at a time, as shown in Figure 12-6.



```
data:
    id:       9
    title:    "Aspernatur ab consequatur nulla et."
    body:     "Magni omnis ut omnis saepe fuga. Ad dolorum porro dolor iure aut nesciunt veniam dolor. Deleniti dolores quis cumque minus repellendus voluptas est. Quos incidunt aspernatur
              quidem sed eum ut ut. Molestiae ratione neque earum ut."
```

Figure 12-6. *JSON response of article number 9*

Notice that for the customized version, you access the model properties directly from the `$this` variable.

How does this work?

This is because a resource class is able to automatically proxy properties and methods, and it accesses the underlying model. Once the resource is defined, it may be returned from a route or controller, as in the `ArticleController` `show` method in the example.

```
//code 12.18
public function show($id)
    {
        //get article
        $article = Article::findOrFail($id);

        //returning single article as a resource
        return new ArticleResource($article);
    }
```

Here you are returning a single article as a resource. Suppose, in case of the `User` resource, you return a new `UserResource` like this:

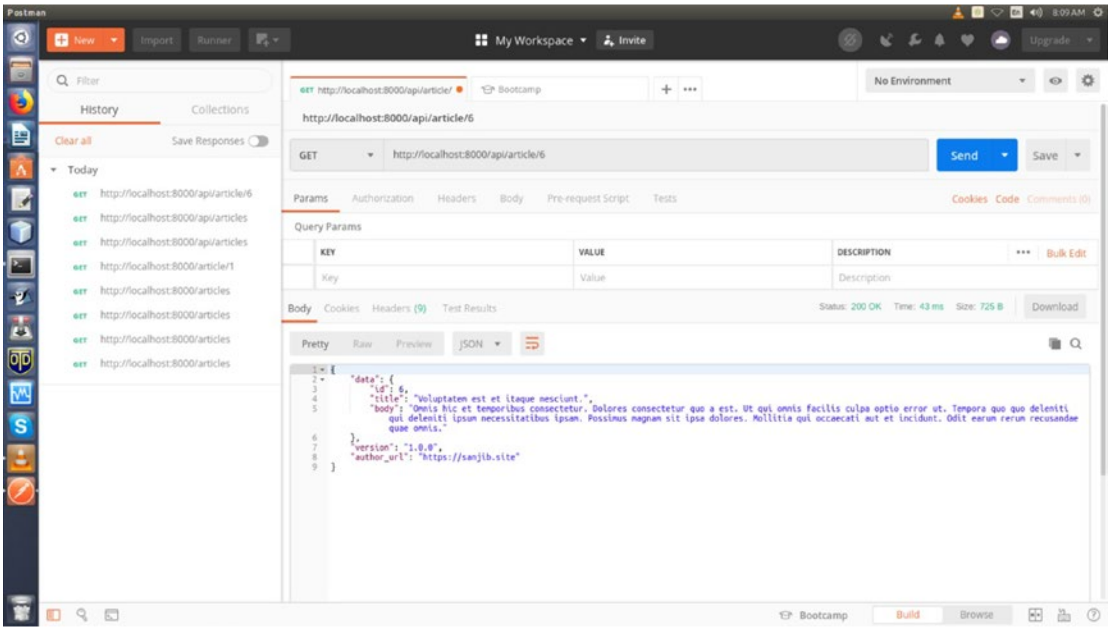
```
//code 12.19
use App\User;
    use App\Http\Resources\User as UserResource;
    Route::get('/user', function () {
        return new UserResource(User::find(1));
    });
```

In the customized version, you can add some more elements with the help of the `with` method in `app/Http/Resources/Article.php`.

The changed code looks like this:

```
//code 12.20
//app/Http/Resources/Article.php
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

class Article extends JsonResource
{
    /**
     * Transform the resource into an array.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return array
     */
    public function toArray($request)
    {
        // return parent::toArray($request);

        return [
            'id' => $this->id,
            'title' => $this->title,
            'body' => $this->body
        ];
    }

    public function with($request) {

        return [
            'version' => '1.0.0',
            'author_url' => url('https://sanjib.site')
        ];

    }
}
```

The JSON response changes to the screen shown in Figure 12-7 in Postman.



```
{
    "data": {
        "id": 6,
        "title": "Voluptatem est et itaque nesciunt.",
        "body": "Omnis hic et temporibus consectetur. Dolores consectetur quo a est. Ut qui omnis facilis culpa optio error ut. Tempora quo quo deleniti
            qui deleniti ipsum necessitatibus ipsam. Possimus magnam sit ipsa dolores. Mollitia qui occaecati aut et incidunt. Odit earum rerum recusandae
            quae omnis."
    },
    "version": "1.0.0",
    "author_url": "https://sanjib.site"
}
```

GET http://localhost:8000/api/article/6
Status: 200 OK　Time: 43 ms　Size: 725 B

Figure 12-7. *You have added the version and author URL link, shown in Postman*

Next, you will see how API collection works. Let's first create a `UserCollection` class.

```
//code 12.21
$ php artisan make:resource UserCollection
Resource collection created successfully.
```

I have added a few dummy users to the application. In the resource `UserCollection`, when you have this line of code:

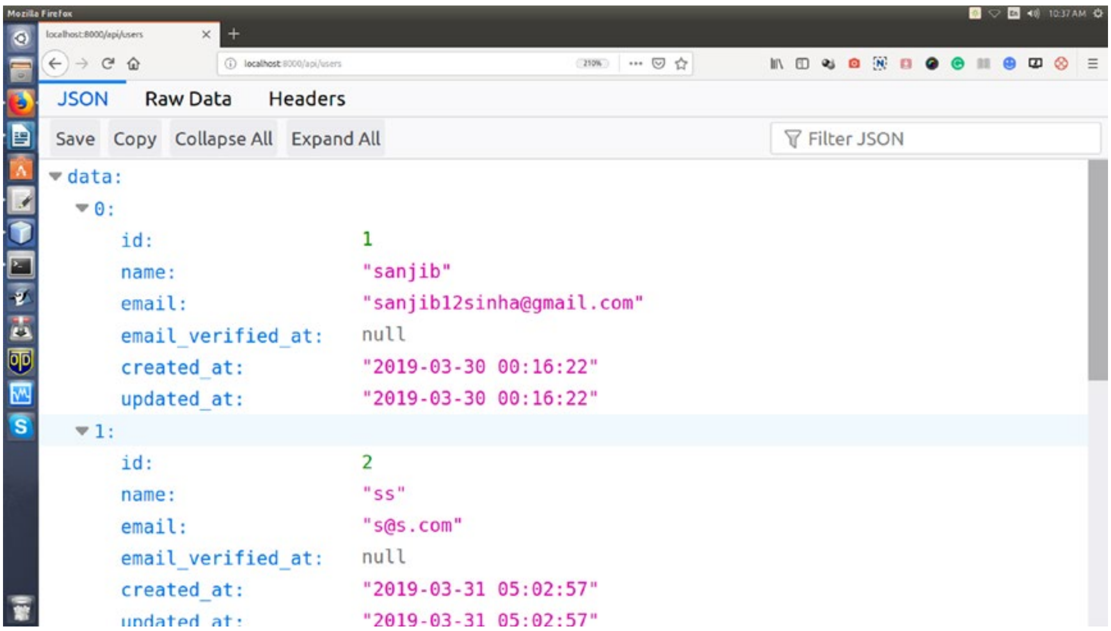
```
//code 12.22
//app/Http/Resources/UserCollection.php
public function toArray($request)
    {
        return parent::toArray($request);
    }
```

you get this JSON response output:

```
//output of code 12.22
data
0
id      1
name    "sanjib"
email   "sanjib12sinha@gmail.com"
email_verified_at    null
created_at   "2019-03-30 00:16:22"
updated_at   "2019-03-30 00:16:22"
1
id      2
name    "ss"
email   "s@s.com"
email_verified_at   null
created_at   "2019-03-31 05:02:57"
updated_at   "2019-03-31 05:02:57"
2
id      3
name    "admin"
email   "admin@la.fr"
email_verified_at   null
created_at   "2019-03-31 05:03:11"
updated_at   "2019-03-31 05:03:11"
```

```
3
id      4
name    "hagudu"
email   "hagudu@hagudu.com"
email_verified_at    null
created_at    "2019-03-31 05:03:24"
updated_at    "2019-03-31 05:03:24"
```

You can take a look at the result in Figure 12-8.



```
data:
  0:
    id:                  1
    name:                "sanjib"
    email:               "sanjib12sinha@gmail.com"
    email_verified_at:   null
    created_at:          "2019-03-30 00:16:22"
    updated_at:          "2019-03-30 00:16:22"
  1:
    id:                  2
    name:                "ss"
    email:               "s@s.com"
    email_verified_at:   null
    created_at:          "2019-03-31 05:02:57"
    updated_at:          "2019-03-31 05:02:57"
```

Figure 12-8. *The usual JSON output of UserCollection*

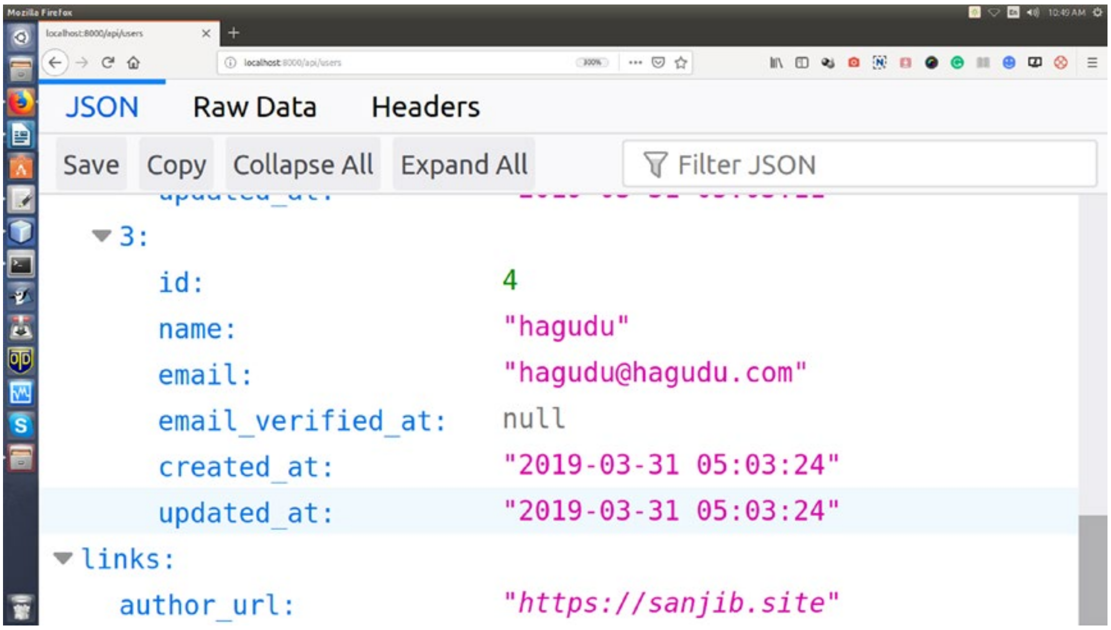
Once the resource collection class has been generated, you can easily define any metadata that should be included with the response. To do that, you need to change the code of `UserCollection` in this way:

```
//code 12.23
//app/Http/Resources/UserCollection.php
    public function toArray($request)
        {
            return [
                'data' => $this->collection,
                'links' => [
                    'author_url' => 'https://sanjib.site',
                ],
            ];
        }
```

Now you have added extra metadata, and in the output it has been included at the bottom of your JSON response.

```
//output of code 12.23
3
id      4
name    "hagudu"
email   "hagudu@hagudu.com"
email_verified_at   null
created_at   "2019-03-31 05:03:24"
updated_at   "2019-03-31 05:03:24"
links
author_url   "https://sanjib.site"
//the code is shortened for brevity
```

In your browser display, it also has been included at the bottom (Figure 12-9).



```
▼ 3:
    id:                   4
    name:                 "hagudu"
    email:                "hagudu@hagudu.com"
    email_verified_at:    null
    created_at:           "2019-03-31 05:03:24"
    updated_at:           "2019-03-31 05:03:24"
▼ links:
    author_url:           "https://sanjib.site"
```

Figure 12-9. *The extra metadata that you have added in the collection*

To get all those new `UserCollection` JSON responses, you can register your route in this way:

```
//code 12.24
//routes/api.php
use App\User;
use App\Http\Resources\UserCollection;

Route::get('/users', function () {
      return new UserCollection(User::all());
});
```

Working with Laravel Passport

You now have authentication in place. As you can see, performing authentication via traditional login forms can easily be done in Laravel.

But API authentication is different. APIs use tokens to authenticate users and do not maintain the session state between requests. Laravel Passport is something that makes developers' lives much easier because it handles the API authentication.

In fact, it does not take much time to implement API authentication when using Laravel Passport. You will see that in a minute. Moreover, you will learn how the OAuth2 server implementation is done, which Laravel Passport also supports.

If you are not familiar with OAuth2 server implementation, then this will be a small introduction for you. OAuth2 is an open standard for access delegation. Many web sites give you a chance to log in via the GitHub, Google, or Facebook APIs. In such cases, GitHub, Amazon, Google, Microsoft, Twitter, and Facebook give permission to those web sites to access their login information without giving them the passwords. This is known as *secured delegated access*. These companies authorize the third-party access to their server resources but never share their credentials.

You are going to do the same thing with a local Laravel application and a remote site called `https://sanjib.site` for demonstration purposes.

To start with, install Passport via the Composer package manager, as shown here:

```
//code 12.25
$ composer require laravel/passport

  - Installing psr/http-message (1.0.1): Downloading (100%)
  - Installing psr/http-factory (1.0.0): Downloading (100%)
  - Installing zendframework/zend-diactoros (2.1.1): Downloading (100%)
  - Installing symfony/psr-http-message-bridge (v1.2.0): Downloading (100%)
  - Installing phpseclib/phpseclib (2.0.15): Downloading (100%)
  - Installing defuse/php-encryption (v2.2.1): Downloading (100%)
  - Installing lcobucci/jwt (3.2.5): Downloading (100%)
  - Installing league/event (2.2.0): Downloading (100%)
  - Installing league/oauth2-server (7.3.3): Downloading (100%)
  - Installing ralouphie/getallheaders (2.0.5): Downloading (100%)
  - Installing guzzlehttp/psr7 (1.5.2): Downloading (100%)
  - Installing guzzlehttp/promises (v1.3.1): Downloading (100%)
  - Installing guzzlehttp/guzzle (6.3.3): Downloading (100%)
  - Installing firebase/php-jwt (v5.0.0): Downloading (100%)
  - Installing laravel/passport (v7.2.2): Downloading (100%)
```

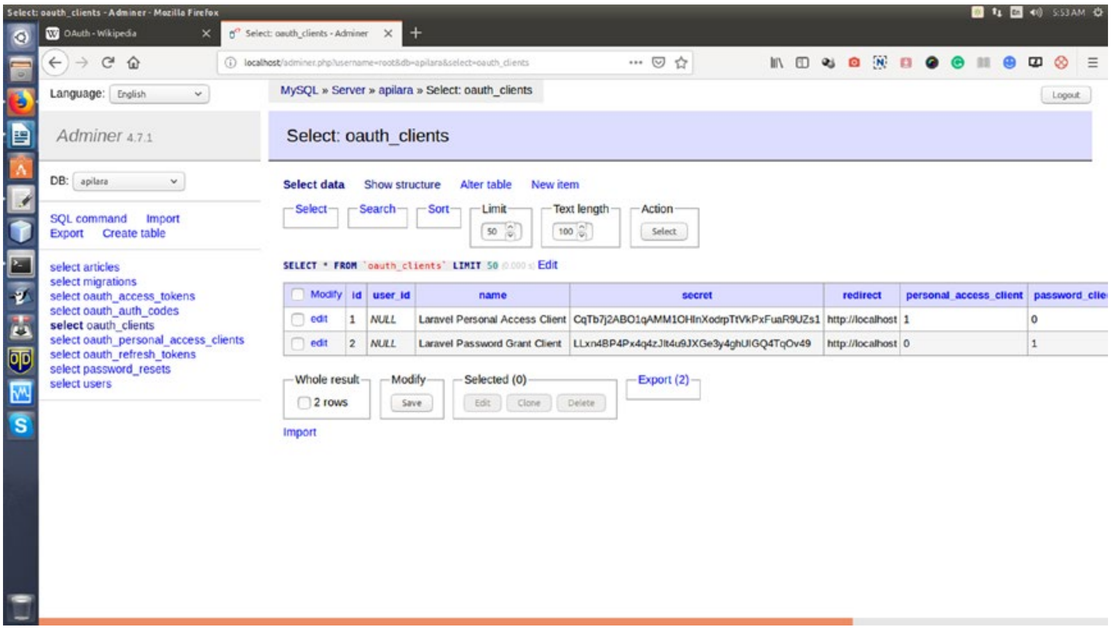
Next, migrate because the Passport service provider registers its own database migration directory with the framework.

```
//code 12.26
$ php artisan migrate
Migrating: 2016_06_01_000001_create_oauth_auth_codes_table
Migrated:  2016_06_01_000001_create_oauth_auth_codes_table
Migrating: 2016_06_01_000002_create_oauth_access_tokens_table
Migrated:  2016_06_01_000002_create_oauth_access_tokens_table
Migrating: 2016_06_01_000003_create_oauth_refresh_tokens_table
Migrated:  2016_06_01_000003_create_oauth_refresh_tokens_table
Migrating: 2016_06_01_000004_create_oauth_clients_table
Migrated:  2016_06_01_000004_create_oauth_clients_table
Migrating: 2016_06_01_000005_create_oauth_personal_access_clients_table
Migrated:  2016_06_01_000005_create_oauth_personal_access_clients_table
```

Now you need to create the encryption keys needed to generate secure access tokens, as shown here:

```
//code 12.27
$ php artisan passport:install
Encryption keys generated successfully.
Personal access client created successfully.
Client ID: 1
Client secret: CqTb7j2ABO1qAMM1OHInXodrpTtVkPxFuaR9UZs1
Password grant client created successfully.
Client ID: 2
Client secret: LLxn4BP4Px4q4zJlt4u9JXGe3y4ghUIGQ4TqOv49
```

As you can see in the previous code, two records, the client ID and the client secret, have been generated (Figure 12-10).



```
SELECT * FROM `oauth_clients` LIMIT 50 (0.000 s) Edit
```

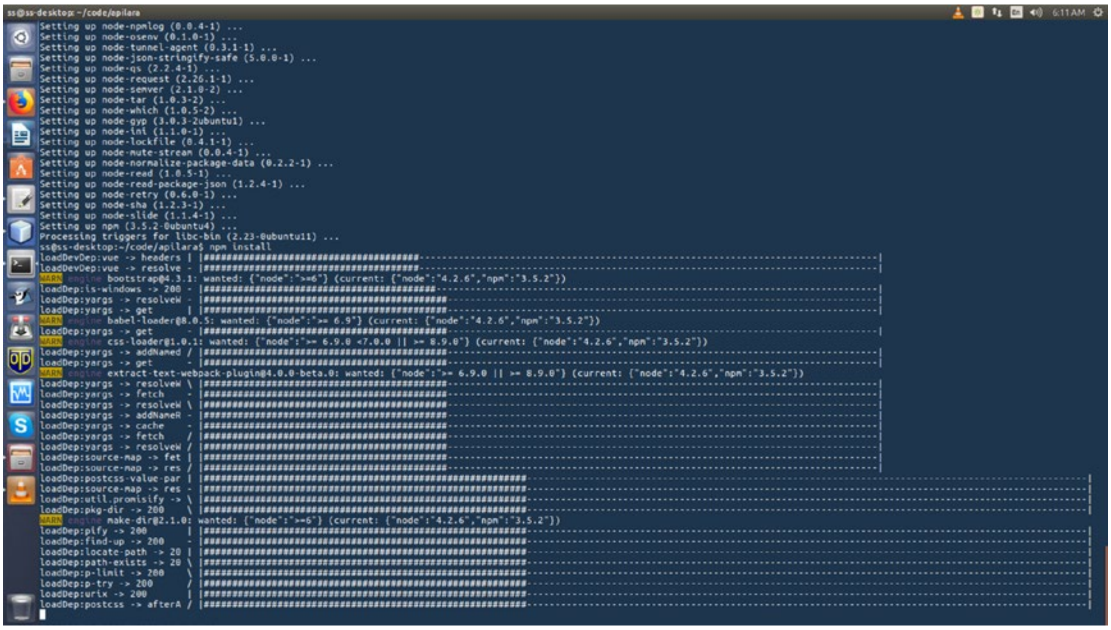
Modify	id	user_id	name	secret	redirect	personal_access_client	password_clie
edit	1	NULL	Laravel Personal Access Client	CqTb7j2ABO1qAMM1OHInXodrpTtVkPxFuaR9UZs1	http://localhost	1	0
edit	2	NULL	Laravel Password Grant Client	LLxn4BP4Px4q4zJlt4u9JXGe3y4ghUIGQ4TqOv49	http://localhost	0	1

Figure 12-10. *oauth_clients in your local database*

At this point, you need to check one more thing: whether you have a recent version of Node.js. If not, you need to install it first by running this command:

```
$ npm install
```

It will take some time for the Node modules to be installed. You can take a look at your terminal while they are being installed (Figure 12-11).



```
ss@ss-desktop: ~/code/apilara
Setting up node-npmlog (0.0.4-1) ...
Setting up node-osenv (0.1.0-1) ...
Setting up node-tunnel-agent (0.3.1-1) ...
Setting up node-json-stringify-safe (5.0.0-1) ...
Setting up node-qs (2.2.4-1) ...
Setting up node-request (2.26.1-1) ...
Setting up node-semver (2.1.0-2) ...
Setting up node-tar (1.0.3-2) ...
Setting up node-which (1.0.5-2) ...
Setting up node-gyp (3.0.3-2ubuntu1) ...
Setting up node-ini (1.1.0-1) ...
Setting up node-lockfile (0.4.1-1) ...
Setting up node-mute-stream (0.0.4-1) ...
Setting up node-normalize-package-data (0.2.2-1) ...
Setting up node-read (1.0.5-1) ...
Setting up node-read-package-json (1.2.4-1) ...
Setting up node-retry (0.6.0-1) ...
Setting up node-sha (1.2.3-1) ...
Setting up node-slide (1.1.4-1) ...
Setting up npm (3.5.2-0ubuntu4) ...
Processing triggers for libc-bin (2.23-0ubuntu11) ...
ss@ss-desktop:~/code/apilara$ npm install
loadDevDep:vue -> headers | |########################################-------------------------------------------------------------------------------------|
loadDevDep:vue -> resolve - |########################################-------------------------------------------------------------------------------------|
WARN engine bootstrap@4.3.1: wanted: {"node":">=6"} (current: {"node":"4.2.6","npm":"3.5.2"})
loadDep:is-windows -> 200 - |###########################################----------------------------------------------------------------------------------|
loadDep:yargs -> resolveW - |############################################---------------------------------------------------------------------------------|
loadDep:yargs -> get      | |############################################---------------------------------------------------------------------------------|
WARN engine babel-loader@8.0.5: wanted: {"node":">= 6.9"} (current: {"node":"4.2.6","npm":"3.5.2"})
loadDep:yargs -> get      - |############################################---------------------------------------------------------------------------------|
WARN engine css-loader@1.0.1: wanted: {"node":">= 6.9.0 <7.0.0 || >= 8.9.0"} (current: {"node":"4.2.6","npm":"3.5.2"})
loadDep:yargs -> addNamed / |############################################---------------------------------------------------------------------------------|
loadDep:yargs -> get      - |############################################---------------------------------------------------------------------------------|
WARN engine extract-text-webpack-plugin@4.0.0-beta.0: wanted: {"node":">= 6.9.0 || >= 8.9.0"} (current: {"node":"4.2.6","npm":"3.5.2"})
loadDep:yargs -> resolveW \ |############################################---------------------------------------------------------------------------------|
loadDep:yargs -> fetch    - |############################################---------------------------------------------------------------------------------|
loadDep:yargs -> resolveW \ |############################################---------------------------------------------------------------------------------|
loadDep:yargs -> addNameR - |############################################---------------------------------------------------------------------------------|
loadDep:yargs -> cache    - |############################################---------------------------------------------------------------------------------|
loadDep:yargs -> fetch    / |############################################---------------------------------------------------------------------------------|
loadDep:yargs -> resolveW / |############################################---------------------------------------------------------------------------------|
loadDep:source-map -> fet | |############################################---------------------------------------------------------------------------------|
loadDep:source-map -> res / |############################################---------------------------------------------------------------------------------|
loadDep:postcss-value-par | |##########################################################----------------------------------------------------------------------------------------------------------|
loadDep:source-map -> res - |##########################################################----------------------------------------------------------------------------------------------------------|
loadDep:util.promisify -> \ |##########################################################----------------------------------------------------------------------------------------------------------|
loadDep:pkg-dir -> 200    \ |##########################################################----------------------------------------------------------------------------------------------------------|
WARN engine make-dir@2.1.0: wanted: {"node":">=6"} (current: {"node":"4.2.6","npm":"3.5.2"})
loadDep:pify -> 200       | |##########################################################----------------------------------------------------------------------------------------------------------|
loadDep:find-up -> 200    - |##########################################################----------------------------------------------------------------------------------------------------------|
loadDep:locate-path -> 20 | |##########################################################----------------------------------------------------------------------------------------------------------|
loadDep:path-exists -> 20 \ |##########################################################----------------------------------------------------------------------------------------------------------|
loadDep:p-limit -> 200    \ |##########################################################----------------------------------------------------------------------------------------------------------|
loadDep:p-try -> 200      / |##########################################################----------------------------------------------------------------------------------------------------------|
loadDep:urix -> 200       | |##########################################################----------------------------------------------------------------------------------------------------------|
loadDep:postcss -> afterA / |##########################################################----------------------------------------------------------------------------------------------------------|
```

Figure 12-11. *The npm install command is running*

Now that you have the Node modules in place, you can tweak some code in your `App\User` model. You need to add the `Laravel\Passport\HasApiTokens` trait in the `User` model. This will be required for your API authentication. Replace your `User` model with this code:

```
//code 12.28
//app/User.php
<?php

namespace App;

use Laravel\Passport\HasApiTokens;
use Illuminate\Notifications\Notifiable;
use Illuminate\Contracts\Auth\MustVerifyEmail;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use HasApiTokens, Notifiable;
    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = [
        'name', 'email', 'password',
    ];

    /**
     * The attributes that should be hidden for arrays.
     *
     * @var array
     */
    protected $hidden = [
        'password', 'remember_token',
    ];

    /**
     * The attributes that should be cast to native types.
     *
     * @var array
     */
    protected $casts = [
        'email_verified_at' => 'datetime',
    ];
}
```

Next, you will call the `Passport::routes` method within the boot method of your `AuthServiceProvider`. This method will register the routes necessary to issue access tokens and revoke access tokens, clients, and personal access tokens. Replace your old code with this:

```
//code 12.29
//app/Providers/ AuthServiceProvider.php
<?php

namespace App\Providers;

use Laravel\Passport\Passport;
use Illuminate\Support\Facades\Gate;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as
ServiceProvider;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * The policy mappings for the application.
     *
     * @var array
     */
    protected $policies = [
        // 'App\Model' => 'App\Policies\ModelPolicy',
    ];

    /**
     * Register any authentication / authorization services.
     *
     * @return void
     */
    public function boot()
    {
        $this->registerPolicies();

        Passport::routes();
    }
}
```

Finally, in your `config/auth.php` configuration file, you should set the `driver` option of the `api` authentication guard to `passport`. This will instruct your application to use Passport's `TokenGuard` when authenticating incoming API requests.

```
//code 12.30
//config/auth.php
'guards' => [
    'web' => [
        'driver' => 'session',
        'provider' => 'users',
    ],
    'api' => [
        'driver' => 'passport',
        'provider' => 'users',
    ],
],
```

Now the time has come to use Vue.js for your application API authentication.

Specifically, you are going to create some Vue components. To do that, you have to publish the Passport Vue components, as shown here:

```
//code 12.31
$ php artisan vendor:publish -tag=passport-components
```

Next, add these lines inside your `resources/js/app.js` file:

```
//code 12.32
//resources/js/app.js
Vue.component(
    'passport-clients',
    require('./components/passport/Clients.vue').default
);

Vue.component(
    'passport-authorized-clients',
    require('./components/passport/AuthorizedClients.vue').default
);

Vue.component(
    'passport-personal-access-tokens',
    require('./components/passport/PersonalAccessTokens.vue').default
);
```

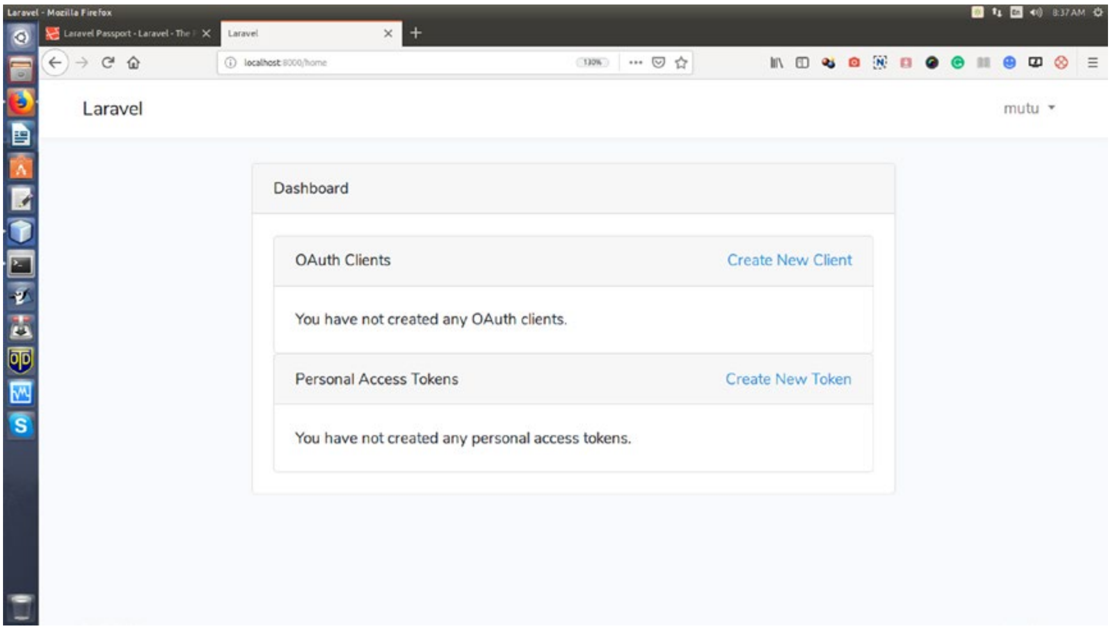
Now that you have your Vue components in the proper place, you can get the template to create the new OAuth clients.

Since you have a traditional login system already in place for your application, you can get that template in your `home.blade.php` file.

```
//code 12.33
//resources/views/home.blade.php
@extends('layouts.app')
```

```
@section('content')
<div class="container">
    <div class="row justify-content-center">
        <div class="col-md-8">
            <div class="card">
                <div class="card-header">Dashboard</div>

                <div class="card-body">
                    @if (session('status'))
                        <div class="alert alert-success" role="alert">
                            {{ session('status') }}
                        </div>
                    @endif

                    <passport-clients></passport-clients>
<passport-authorized-clients></passport-authorized-clients>
<passport-personal-access-tokens></passport-personal-access-tokens>
                </div>
            </div>
        </div>
    </div>
</div>
@endsection
```

Now you are ready to have your users create their own OAuth API clients. Once a user is signed in, they will be greeted by the new look shown in Figure 12-12.



Laravel

mutu

Dashboard

OAuth Clients   Create New Client

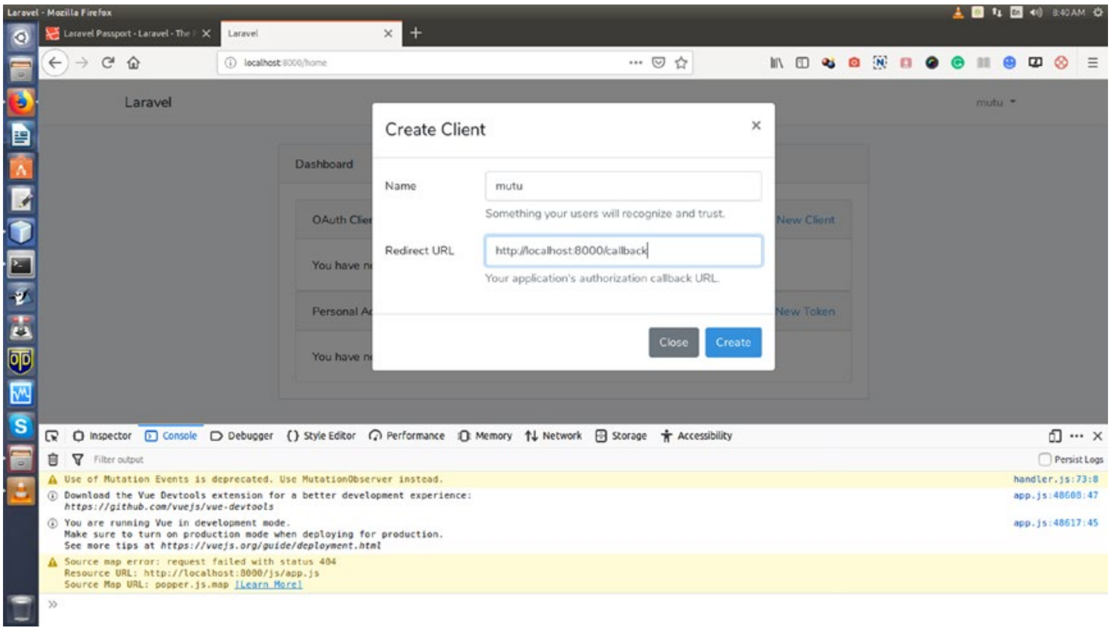
You have not created any OAuth clients.

Personal Access Tokens   Create New Token

You have not created any personal access tokens.

Figure 12-12. *The user has not created any client yet*

Now the user is going to create the first OAuth client (Figure 12-13).



Create Client

Name: mutu

Something your users will recognize and trust.

Redirect URL: http://localhost:8000/callback

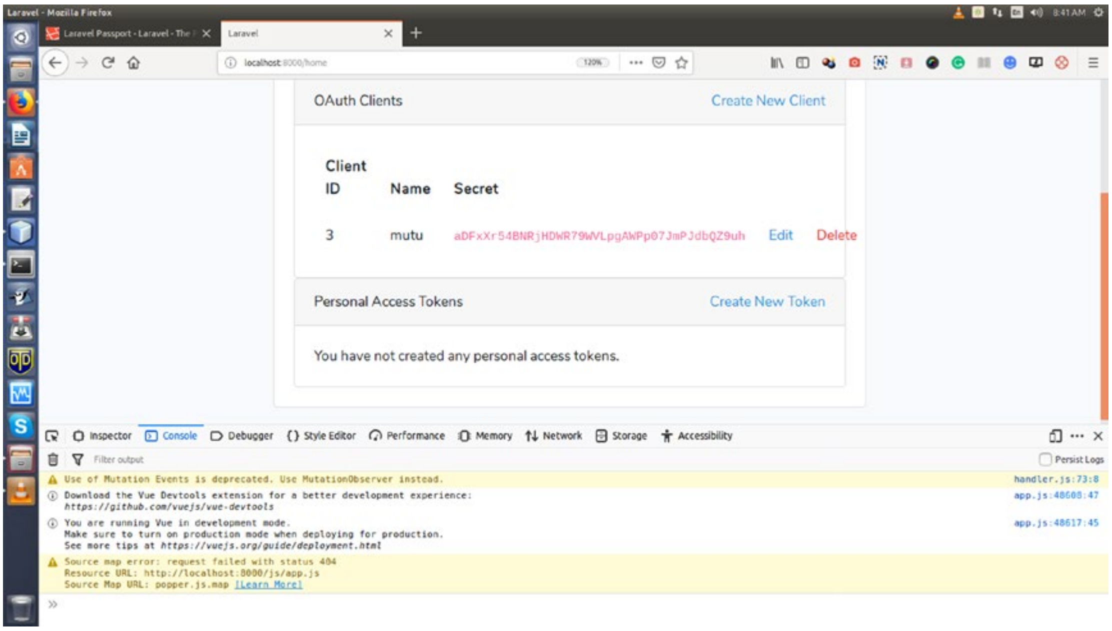
Your application's authorization callback URL.

Close   Create

```
Use of Mutation Events is deprecated. Use MutationObserver instead.        handler.js:73:8
Download the Vue Devtools extension for a better development experience:   app.js:48608:47
https://github.com/vuejs/vue-devtools
You are running Vue in development mode.                                    app.js:48617:45
Make sure to turn on production mode when deploying for production.
See more tips at https://vuejs.org/guide/deployment.html
Source map error: request failed with status 404
Resource URL: http://localhost:8000/js/app.js
Source Map URL: popper.js.map [Learn More]
```

Figure 12-13. *The user is going to create the new OAuth client*

After the first OAuth client has been created, it will immediately be reflected on the page (Figure 12-14).



OAuth Clients

Client ID	Name	Secret		
3	mutu	aDFxXr54BNRjHDWR79WVLpgAWPp07JmPJdbQZ9uh	Edit	Delete

Personal Access Tokens

You have not created any personal access tokens.

```
Use of Mutation Events is deprecated. Use MutationObserver instead.          handler.js:73:8
Download the Vue Devtools extension for a better development experience:     app.js:48608:47
https://github.com/vuejs/vue-devtools
You are running Vue in development mode.                                     app.js:48617:45
Make sure to turn on production mode when deploying for production.
See more tips at https://vuejs.org/guide/deployment.html
Source map error: request failed with status 404
Resource URL: http://localhost:8000/js/app.js
Source Map URL: popper.js.map [Learn More]
```

Figure 12-14. *The first OAuth client has been created*

Now you have come to the final stage, which is API authentication.

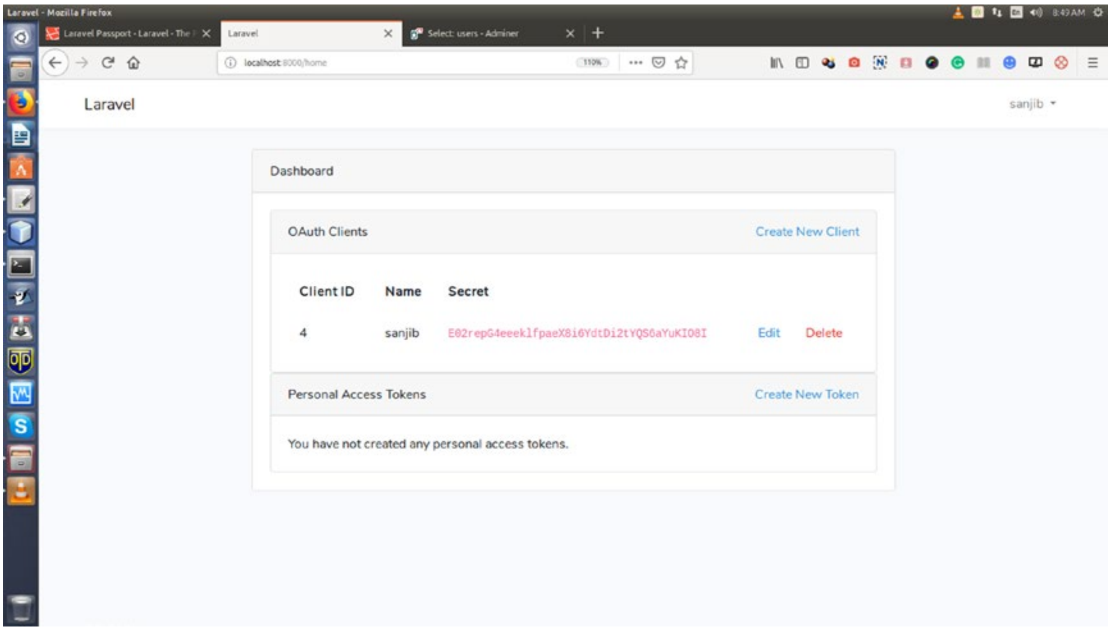
API Authentication

You have seen how a user has added an OAuth client on their own home page. But a user with administrative privileges can add many clients for other users. Suppose you have a user who has an ID of 2.

In that case, the administrator can issue the following commands in the terminal to create the OAuth client for that user:

```
//code 12.34
$ php artisan passport:client

 Which user ID should the client be assigned to?:
 > 2
 What should we name the client?:
 > sanjib

 Where should we redirect the request after authorization? [http://
localhost/auth/callback]:
 >

New client created successfully.
Client ID: 4
Client secret: E02repG4eeeklfpaeX8i6YdtDi2tYQS6aYuKI08I
```

Immediately after this addition, the user with ID 2 will see the OAuth client on their `home.blade.php` page (Figure 12-15).



Dashboard

OAuth Clients — Create New Client

Client ID	Name	Secret		
4	sanjib	E02repG4eeeklfpaeX8i6YdtDi2tYQS6aYuKI08I	Edit	Delete

Personal Access Tokens — Create New Token

You have not created any personal access tokens.

Figure 12-15. *A new OAuth client has been added by the administrator*

Now what happens if a remote site (here `https://sanjib.site`) wants your application to authorize user 5?

You will be able to authorize it without giving your login credentials.

To do that, in the routes/web.php file of https://sanjib.site, there is this code:

```
//code 12.35
Route::get('/redirect', function () {
    $query = http_build_query([
        'client_id' => '5',
        'redirect_uri' => 'https://sanjib.site/callback',
        'response_type' => 'code',
        'scope' => '',
    ]);

    return redirect('http://localhost:8000/oauth/authorize?'.$query);
});
```

This will redirect the page to the login screen of http://localhost:8000/login, as shown in Figure 12-16.

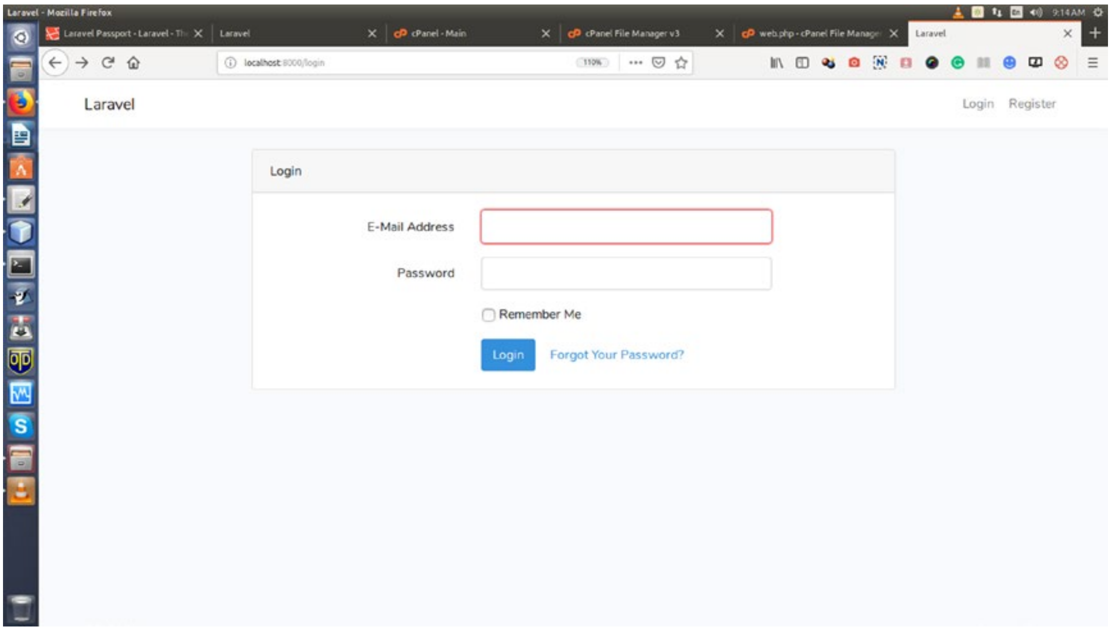
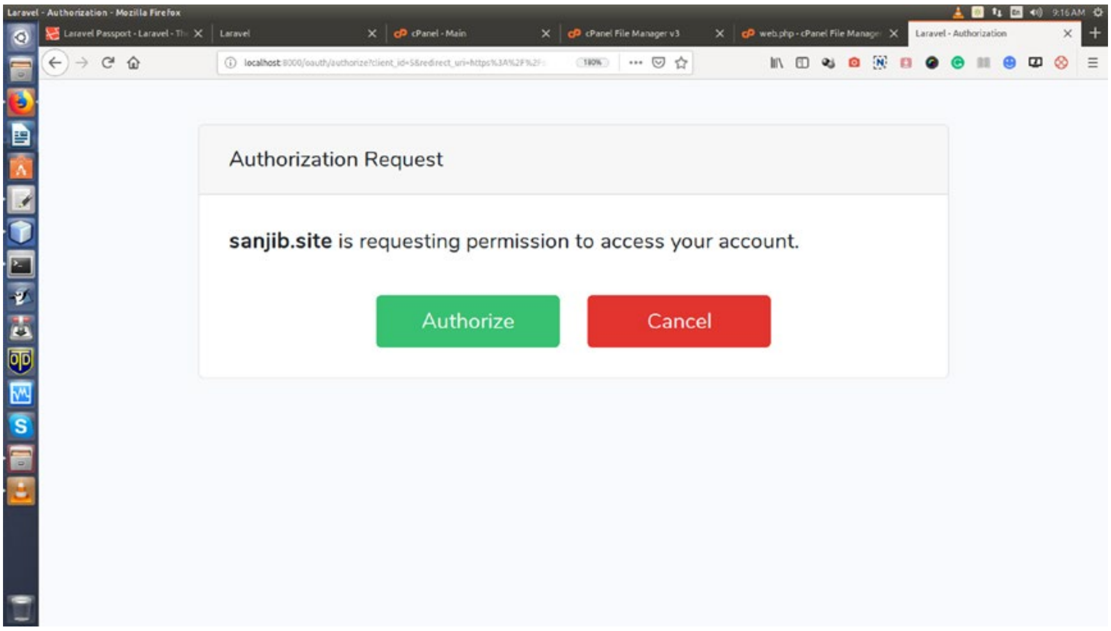


Figure 12-16. *Redirected to the login screen of http://localhost:8000/login*

Remember, the remote site (https://sanjib.site) asks for your permission to allow the user with ID 5. This happens because this route has already been registered in the web.php file of https://sanjib.site.

For that reason, only the fifth user can view this login page. If some other users want to log in, an error will be thrown.

However, once the fifth user logs in, they will be greeted with the page in Figure 12-17.


Authorization Request

sanjib.site is requesting permission to access your account.

Figure 12-17. *sanjib.site is requesting permission to access the account of the user with an ID of 5*

Now, if you look at the URL in the browser, you will see something similar to this:

```
//code 12.36
http://localhost:8000/oauth/authorize?client_id=5&redirect_
uri=https%3A%2F%2Fsanjib.site%2Fcallback&response_type=code&scope=
```

Laravel APIs, Passport, and API authentication features are great additions to easily manage the difficulties of authenticating the OAuth client.

I hope you have an idea of how you could use these great features in your application.

APPENDIX

More New Features in Laravel 5.8

Laravel 5.8 has many improvements and updates. Some of them seem to be quite interesting and will impact the course of future development. In this appendix, you'll take a quick look at some of them.

One of the most interesting changes is the "dump server" feature. It was first incorporated in Laravel 5.7 and then extended in Laravel 5.8.

What Is the Dump Server Feature?

The dump server feature allows you to start a "dump server" to collect dump information about the internal processes.

Let's start a new version of Laravel 5.8 installed locally. Then open the `composer.json` file and take a look at this part:

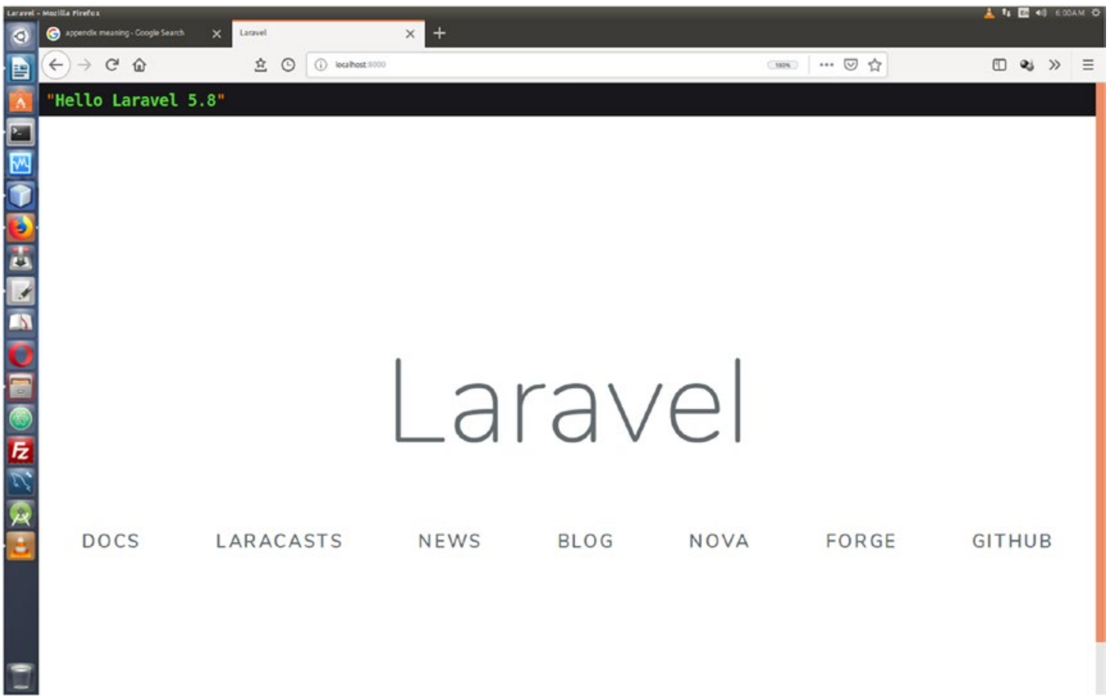
```
"require-dev": {
    "beyondcode/laravel-dump-server": "^1.0",
    "filp/whoops": "^2.0",
    "fzaninotto/faker": "^1.4",
    "mockery/mockery": "^1.0",
    "nunomaduro/collision": "^2.0",
    "phpunit/phpunit": "^7.5"
},
```

You'll see `beyondcode/laravel-dump-server": "^1.0"` already in place.

Next, open the `routes/web.php` file and add this line:

```
Route::get('/', function () {

    dump("Hello Laravel 5.8");

    return view('welcome');
});
```

This gives you the output shown in Figure A-1.



```
"Hello Laravel 5.8"
```

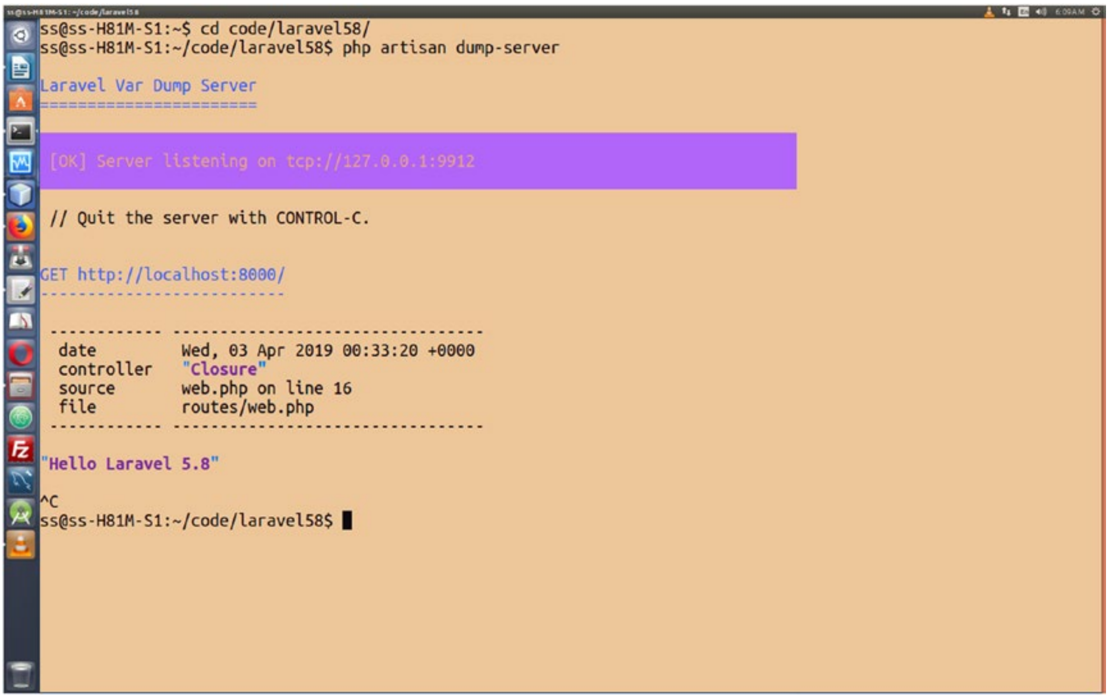
Figure A-1. *Dump server output in the browser*

Next, in the terminal, so ahead and pass this command:

```
$ php artisan dump-server
It gives us this output:
ss@ss-H81M-S1:~$ cd code/laravel58/
ss@ss-H81M-S1:~/code/laravel58$ php artisan dump-server
```

```
Laravel Var Dump Server
=======================

 [OK] Server listening on tcp://127.0.0.1:9912

 // Quit the server with CONTROL-C.

GET http://localhost:8000/
--------------------------

 ------------ ---------------------------------
  date         Wed, 03 Apr 2019 00:33:20 +0000
  controller   "Closure"
  source       web.php on line 16
  file         routes/web.php
 ------------ ---------------------------------

"Hello Laravel 5.8"
```

Figure A-2 shows the output in the terminal. It is interesting to take note that in the browser, at the same time, the page just vanishes.



```
ss@ss-H81M-S1:~$ cd code/laravel58/
ss@ss-H81M-S1:~/code/laravel58$ php artisan dump-server

Laravel Var Dump Server
=======================

 [OK] Server listening on tcp://127.0.0.1:9912

 // Quit the server with CONTROL-C.

GET http://localhost:8000/
--------------------------

 ----------- ------------------------------
  date        Wed, 03 Apr 2019 00:33:20 +0000
  controller  "Closure"
  source      web.php on line 16
  file        routes/web.php
 ----------- ------------------------------

"Hello Laravel 5.8"

^C
ss@ss-H81M-S1:~/code/laravel58$
```

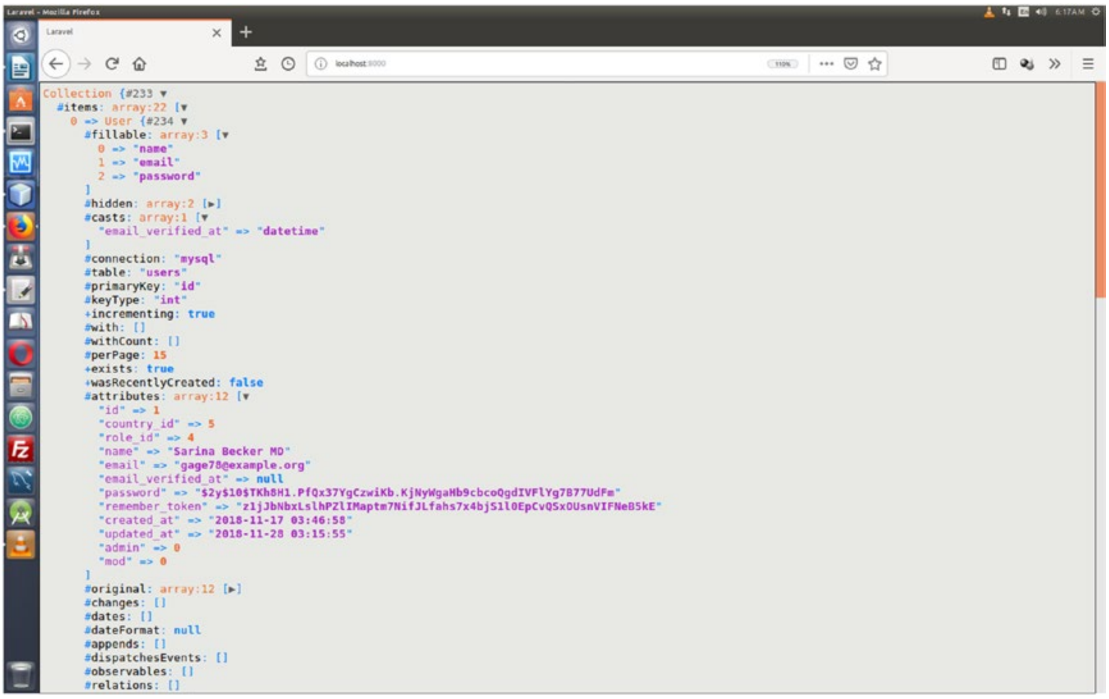
Figure A-2. *Dump server output in the terminal*

Let's dump all the users the same way. Remember, I have seeded the `users` table with fake data, and currently there are 21 users in the database.

Go ahead and change your `routes/web.php` file in this way:

```
use App\User;

Route::get('/', function () {

    $users = User::all();

    dump($users);

    return view('welcome');
});
```

Open your browser, and you will see something like Figure A-3.



```
Collection {#233 ▼
  #items: array:22 [▼
    0 => User {#234 ▼
      #fillable: array:3 [▼
        0 => "name"
        1 => "email"
        2 => "password"
      ]
      #hidden: array:2 [▶]
      #casts: array:1 [▼
        "email_verified_at" => "datetime"
      ]
      #connection: "mysql"
      #table: "users"
      #primaryKey: "id"
      #keyType: "int"
      +incrementing: true
      #with: []
      #withCount: []
      #perPage: 15
      +exists: true
      +wasRecentlyCreated: false
      #attributes: array:12 [▼
        "id" => 1
        "country_id" => 5
        "role_id" => 4
        "name" => "Sarina Becker MD"
        "email" => "gage78@example.org"
        "email_verified_at" => null
        "password" => "$2y$10$TKh8H1.PfQx37YgCzwiKb.KjNyWgaMb9cbcoQgdIVFlYg7B77UdFm"
        "remember_token" => "z1jJbNbxLslhPZlIMaptm7NifJLfahs7x4bjS1l0EpCvQSxOUsnVIFNeB5kE"
        "created_at" => "2018-11-17 03:46:58"
        "updated_at" => "2018-11-28 03:15:55"
        "admin" => 0
        "mod" => 0
      ]
      #original: array:12 [▶]
      #changes: []
      #dates: []
      #dateFormat: null
      #appends: []
      #dispatchesEvents: []
      #observables: []
      #relations: []
```

Figure A-3. *The browser is filled with the dumped user data*

However, you can manage this output by running the `dump-server artisan` command in the terminal. You will then get the clean output in the browser (Figure A-4).

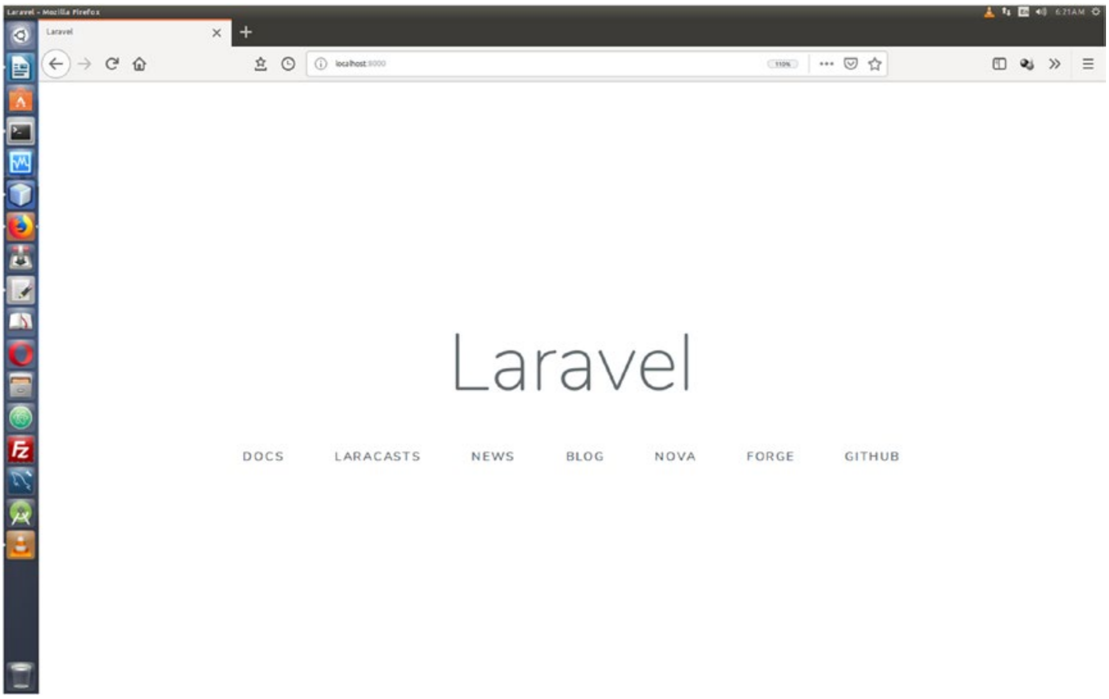
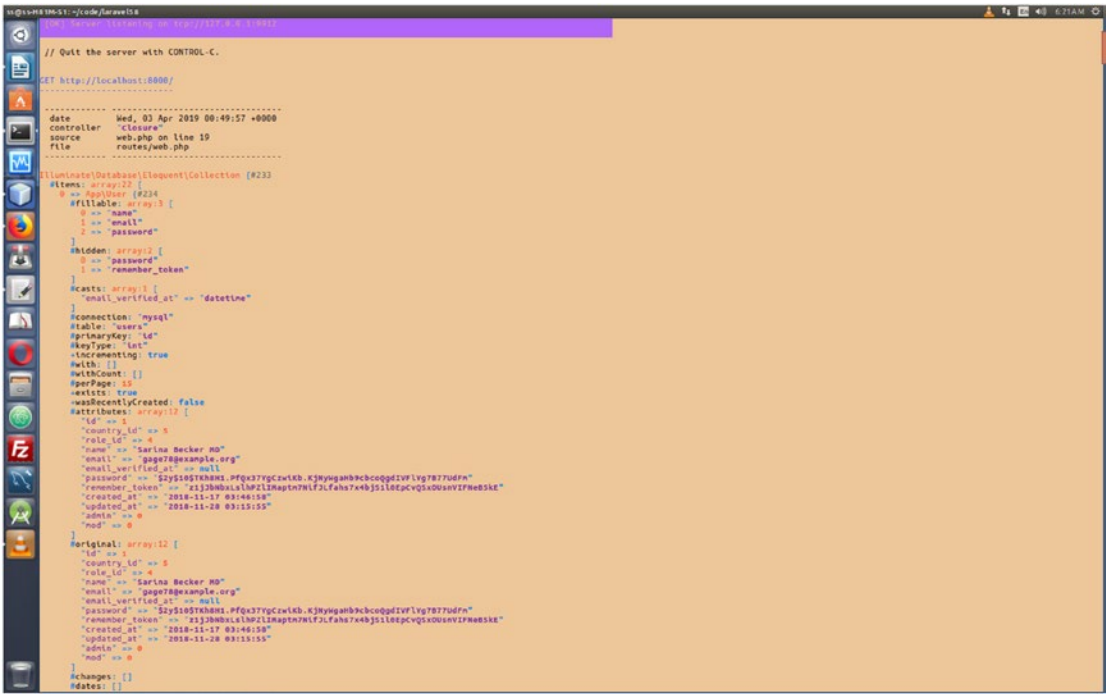


Figure A-4. *Clear browser without dumped user data*

Instead, all the user data is being displayed in the terminal, as shown in Figure A-5.



```
ss@ss-H81M-S1: ~/code/laravel58
[OK] Server listening on tcp://127.0.0.1:9912

// Quit the server with CONTROL-C.

GET http://localhost:8000/
--------------------------

 ------------ ----------------------------------
  date         Wed, 03 Apr 2019 00:49:57 +0000
  controller   "Closure"
  source       web.php on line 19
  file         routes/web.php
 ------------ ----------------------------------

Illuminate\Database\Eloquent\Collection {#233
  #items: array:22 [
    0 => App\User {#234
      #fillable: array:3 [
        0 => "name"
        1 => "email"
        2 => "password"
      ]
      #hidden: array:2 [
        0 => "password"
        1 => "remember_token"
      ]
      #casts: array:1 [
        "email_verified_at" => "datetime"
      ]
      #connection: "mysql"
      #table: "users"
      #primaryKey: "id"
      #keyType: "int"
      +incrementing: true
      #with: []
      #withCount: []
      #perPage: 15
      +exists: true
      +wasRecentlyCreated: false
      #attributes: array:12 [
        "id" => 1
        "country_id" => 5
        "role_id" => 4
        "name" => "Sarina Becker MD"
        "email" => "gage78@example.org"
        "email_verified_at" => null
        "password" => "$2y$10$TKh8H1.PfQx37YgCzwiKb.KjNyHgaHb9cbcoQgdIVFlVg7B77UdFm"
        "remember_token" => "z1jJbNbxLslhPZlIMaptm7NifJLfahs7x4bj51l0EpCvQ5xOUsnVIFNeB5kE"
        "created_at" => "2018-11-17 03:46:58"
        "updated_at" => "2018-11-28 03:15:55"
        "admin" => 0
        "mod" => 0
      ]
      #original: array:12 [
        "id" => 1
        "country_id" => 5
        "role_id" => 4
        "name" => "Sarina Becker MD"
        "email" => "gage78@example.org"
        "email_verified_at" => null
        "password" => "$2y$10$TKh8H1.PfQx37YgCzwiKb.KjNyHgaHb9cbcoQgdIVFlVg7B77UdFm"
        "remember_token" => "z1jJbNbxLslhPZlIMaptm7NifJLfahs7x4bj51l0EpCvQ5xOUsnVIFNeB5kE"
        "created_at" => "2018-11-17 03:46:58"
        "updated_at" => "2018-11-28 03:15:55"
        "admin" => 0
        "mod" => 0
      ]
      #changes: []
      #dates: []
```

Figure A-5. *Dumped user data in the terminal*

Now change the `routes/web.php` code to the following and run the `dump-server` command again:

```
use App\User;

Route::get('/', function () {

    $users = User::all()->toArray();

    dump($users);

    return view('welcome');
});
```

The output is dumped in the terminal like in Figure A-6.

```
ss@ss-H81M-S1:~/code/laravel58$ php artisan dump-server

Laravel Var Dump Server
=======================
 [OK] Server listening on tcp://127.0.0.1:9912
 // Quit the server with CONTROL-C.
GET http://localhost:8000/
--------------------------
 ------------ ---------------------------------
  date         Wed, 03 Apr 2019 00:56:02 +0000
  controller   "Closure"
  source       web.php on line 19
  file         routes/web.php
 ------------ ---------------------------------
array:22 [
  0 => array:10 [
    "id" => 1
    "country_id" => 5
    "role_id" => 4
    "name" => "Sarina Becker MD"
    "email" => "gage78@example.org"
    "email_verified_at" => null
    "created_at" => "2018-11-17 03:46:58"
    "updated_at" => "2018-11-28 03:15:55"
    "admin" => 0
    "mod" => 0
  ]
  1 => array:10 [
    "id" => 2
    "country_id" => 7
    "role_id" => null
    "name" => "Ariel Hand"
    "email" => "mallie.treutel@example.org"
    "email_verified_at" => null
    "created_at" => "2018-11-17 03:46:59"
    "updated_at" => "2018-11-17 03:46:59"
    "admin" => 0
    "mod" => 0
  ]
  2 => array:10 [
    "id" => 3
    "country_id" => null
    "role_id" => null
    "name" => "Carissa Raynor"
    "email" => "kuhic.eliezer@example.com"
    "email_verified_at" => null
    "created_at" => "2018-11-17 03:46:59"
    "updated_at" => "2018-11-17 03:46:59"
    "admin" => 0
    "mod" => 0
  ]
  3 => array:10 [
    "id" => 4
    "country_id" => null
    "role_id" => null
    "name" => "Jude Johnston"
    "email" => "murazik.devante@example.com"
    "email_verified_at" => null
    "created_at" => "2018-11-17 03:46:59"
    "updated_at" => "2018-11-17 03:46:59"
    "admin" => 0
    "mod" => 0
  ]
  4 => array:10 [
    "id" => 5
    "country_id" => null
    "role_id" => null
    "name" => "Sarai Beier"
    "email" => "rowe.lamont@example.net"
    "email_verified_at" => null
    "created_at" => "2018-11-17 03:46:59"
    "updated_at" => "2018-11-17 03:46:59"
    "admin" => 0
    "mod" => 0
  ]
  5 => array:10 [
    "id" => 6
    "country_id" => null
    "role_id" => null
    "name" => "Ms. Freda Kemmer"
    "email" => "brock56@example.org"
    "email_verified_at" => null
    "created_at" => "2018-11-17 03:46:59"
    "updated_at" => "2018-11-17 03:46:59"
    "admin" => 0
    "mod" => 0
  ]
```

I have omitted some output here for brevity.

Improved artisan Command

Laravel 5.8 now has improved `artisan` commands. Before Laravel 5.8, you could not run two servers in parallel on different ports. For example, suppose you are running your application in Laravel 5.7 on port 8000. At the same time, you want to run your Laravel 5.8 application on another port.

With previous versions, Laravel ran the servers on one port, 8000. Laravel 5.8 has the ability to use another port.

Let's see an example. I have a news application in my `code/news` directory. It runs on Laravel 5.7. I start it, and along with it, I start a new Laravel 5.8 application in the `code/laravel58` directory.

The first `artisan serve` command has normal output like this:

```
//output of Laravel 5.7 php artisan serve command
cdss@ss-H81M-S1:~$ cd code/news/
ss@ss-H81M-S1:~/code/news$ php artisan serve
Laravel development server started: <http://127.0.0.1:8000>
```

```
[Wed Apr  3 08:22:26 2019] 127.0.0.1:40984 [200]: /css/webmag/css/
                                                  bootstrap.min.css
[Wed Apr  3 08:22:26 2019] 127.0.0.1:40986 [200]: /css/webmag/css/font-
                                                  awesome.min.css
[Wed Apr  3 08:22:26 2019] 127.0.0.1:40988 [200]: /css/webmag/css/style.css
[Wed Apr  3 08:22:26 2019] 127.0.0.1:40990 [200]: /css/webmag/js/jquery.
                                                  min.js
[Wed Apr  3 08:22:26 2019] 127.0.0.1:40992 [200]: /css/webmag/js/bootstrap.
                                                  min.js
[Wed Apr  3 08:22:26 2019] 127.0.0.1:40994 [200]: /css/webmag/js/main.js
[Wed Apr  3 08:22:26 2019] 127.0.0.1:40996 [200]: /css/webmag/img/logo.png
[Wed Apr  3 08:22:27 2019] 127.0.0.1:41004 [200]: /images/entertainment/
                                                  rowan-chestnut-175871-
                                                  unsplash.jpg
[Wed Apr  3 08:22:27 2019] 127.0.0.1:41006 [200]: /css/webmag/img/
                                                  widget-1.jpg
(Code is incomplete for brevity)
```

Next, I run the new Laravel 5.8 application in `code/laravel58`. Take a look at the output in the terminal to see the difference, as shown here:

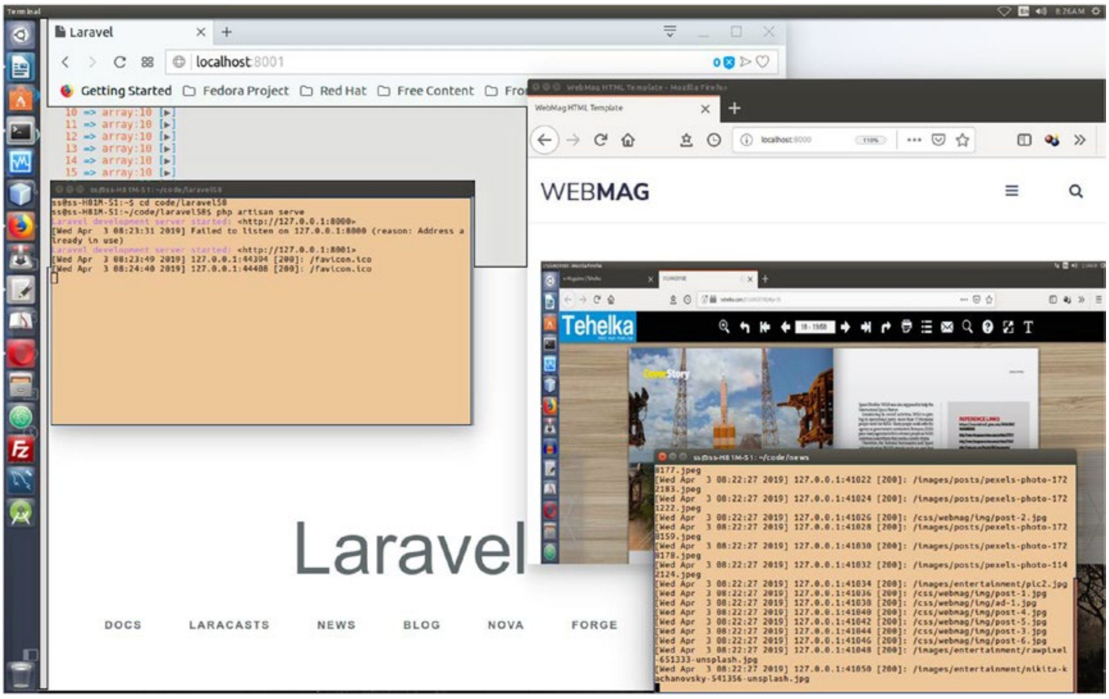
```
//output of Laravel 5.8 php artisan serve command
ss@ss-H81M-S1:~$ cd code/laravel58
ss@ss-H81M-S1:~/code/laravel58$ php artisan serve
Laravel development server started: <http://127.0.0.1:8000>
[Wed Apr  3 08:23:31 2019] Failed to listen on 127.0.0.1:8000 (reason:
Address already in use)
Laravel development server started: <http://127.0.0.1:8001>
[Wed Apr  3 08:23:49 2019] 127.0.0.1:44394 [200]: /favicon.ico
[Wed Apr  3 08:24:40 2019] 127.0.0.1:44408 [200]: /favicon.ico
```

In the previous code, these two lines are important:

```
[Wed Apr  3 08:23:31 2019] Failed to listen on 127.0.0.1:8000 (reason:
Address already in use)
Laravel development server started: <http://127.0.0.1:8001>
```

Since port 8000 is already in use, Laravel 5.8 is smart enough to start another server on port 8001. According to the new features of Laravel 5.8, it will scan up to port 8002 and try to find a port that is free, as shown in Figure A-6.

This is a great advancement because now locally you can run multiple applications at the same time.



```
10 => array:10 [▶]
11 => array:10 [▶]
12 => array:10 [▶]
13 => array:10 [▶]
14 => array:10 [▶]
15 => array:10 [▶]
```

```
ss@ss-H81M-S1:~$ cd code/laravel58
ss@ss-H81M-S1:~/code/laravel58$ php artisan serve
Laravel development server started: <http://127.0.0.1:8000>
[Wed Apr  3 08:23:31 2019] Failed to listen on 127.0.0.1:8000 (reason: Address a
lready in use)
Laravel development server started: <http://127.0.0.1:8001>
[Wed Apr  3 08:23:49 2019] 127.0.0.1:44394 [200]: /favicon.ico
[Wed Apr  3 08:24:40 2019] 127.0.0.1:44408 [200]: /favicon.ico
```

```
8177.jpeg
[Wed Apr  3 08:22:27 2019] 127.0.0.1:41022 [200]: /images/posts/pexels-photo-172
2183.jpeg
[Wed Apr  3 08:22:27 2019] 127.0.0.1:41024 [200]: /images/posts/pexels-photo-172
1222.jpeg
[Wed Apr  3 08:22:27 2019] 127.0.0.1:41026 [200]: /css/webmag/img/post-2.jpg
[Wed Apr  3 08:22:27 2019] 127.0.0.1:41028 [200]: /images/posts/pexels-photo-172
8159.jpeg
[Wed Apr  3 08:22:27 2019] 127.0.0.1:41030 [200]: /images/posts/pexels-photo-172
8178.jpeg
[Wed Apr  3 08:22:27 2019] 127.0.0.1:41032 [200]: /images/posts/pexels-photo-114
2124.jpeg
[Wed Apr  3 08:22:27 2019] 127.0.0.1:41034 [200]: /images/entertainment/pic2.jpg
[Wed Apr  3 08:22:27 2019] 127.0.0.1:41036 [200]: /css/webmag/img/post-1.jpg
[Wed Apr  3 08:22:27 2019] 127.0.0.1:41038 [200]: /css/webmag/img/ad-1.jpg
[Wed Apr  3 08:22:27 2019] 127.0.0.1:41040 [200]: /css/webmag/img/post-4.jpg
[Wed Apr  3 08:22:27 2019] 127.0.0.1:41042 [200]: /css/webmag/img/post-5.jpg
[Wed Apr  3 08:22:27 2019] 127.0.0.1:41044 [200]: /css/webmag/img/post-3.jpg
[Wed Apr  3 08:22:27 2019] 127.0.0.1:41046 [200]: /css/webmag/img/post-6.jpg
[Wed Apr  3 08:22:27 2019] 127.0.0.1:41048 [200]: /images/entertainment/rawpixel
-651333-unsplash.jpg
[Wed Apr  3 08:22:27 2019] 127.0.0.1:41050 [200]: /images/entertainment/nikita-k
achanovsky-541356-unsplash.jpg
```

***Figure A-6.** Along with two terminals, two browsers running in parallel on two ports*

There is another improvement in the form of `Artisan::call`. You use the `Artisan::call` method when you want to call the method programmatically. In the past, you could pass some options to the command this way:

```
Artisan::call('migrate:install', ['database' => 'laravelpractice']);
```

But with Laravel 5.8, this has changed drastically. Now Laravel imports the console environment and uses flags just like `artisan` commands in the terminal.

```
Artisan::call('migrate:install -database=laravelpractice');
```

A Few More Additions

The following are a few other additions.

Renaming the Mail Format Folder

You learned about e-mail verification in Chapter 10. The e-mail validation methods have been improved a lot. This is especially true when you have mailable classes in your project, as you saw in Chapter 12. I usually customize the components with the `vendor:publish artisan` command.

In earlier versions, the created directory would be named `/resources/views/vendor/mail/markdown`; it is now named `/resources/views/vendor/mail/text`. The logic behind this is simple but meaningful. Both folders can contain Markdown code for making good-looking, responsive HTML templates with plain-text fallbacks. That is why the `markdown` folder has been renamed to `text`.

Changes to .env

A major change has taken place in the `.env` file. Laravel 5.8 uses the relatively new DOTENV 3.0 to manage the `.env` file. It will be an extremely important change in Laravel application development in the future.

Why? Let me explain. Until now, Laravel's `.env` does not support multiple lines and whitespace. The new features of DOTENV 3.0 support multiline strings. So, inside the `.env` file, you can write like this:

```
DEVELOPMENT_APP_KEY="multiline
strings"
```

Only the first line would have been parsed before. Now the whole string will be parsed. That means when implementing security in the future, you can now ask for multiline API keys.

Changing the Look of Error Pages

There has been a big change in how error pages look. Figure A-7 shows an error page in a Laravel 5.7 application.

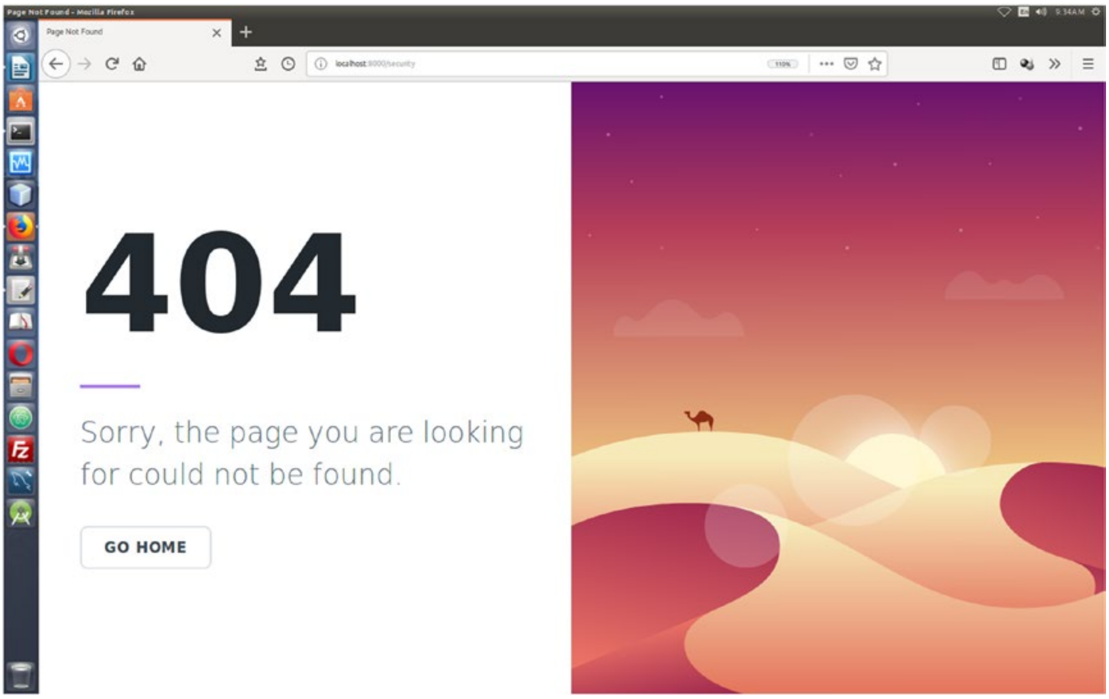

Page Not Found - Mozilla Firefox
Page Not Found
localhost:8000/security
404
Sorry, the page you are looking for could not be found.
GO HOME

Figure A-7. *Error page in Laravel 5.7*

Some say that the new look is more modern, although you may disagree (Figure A-8).

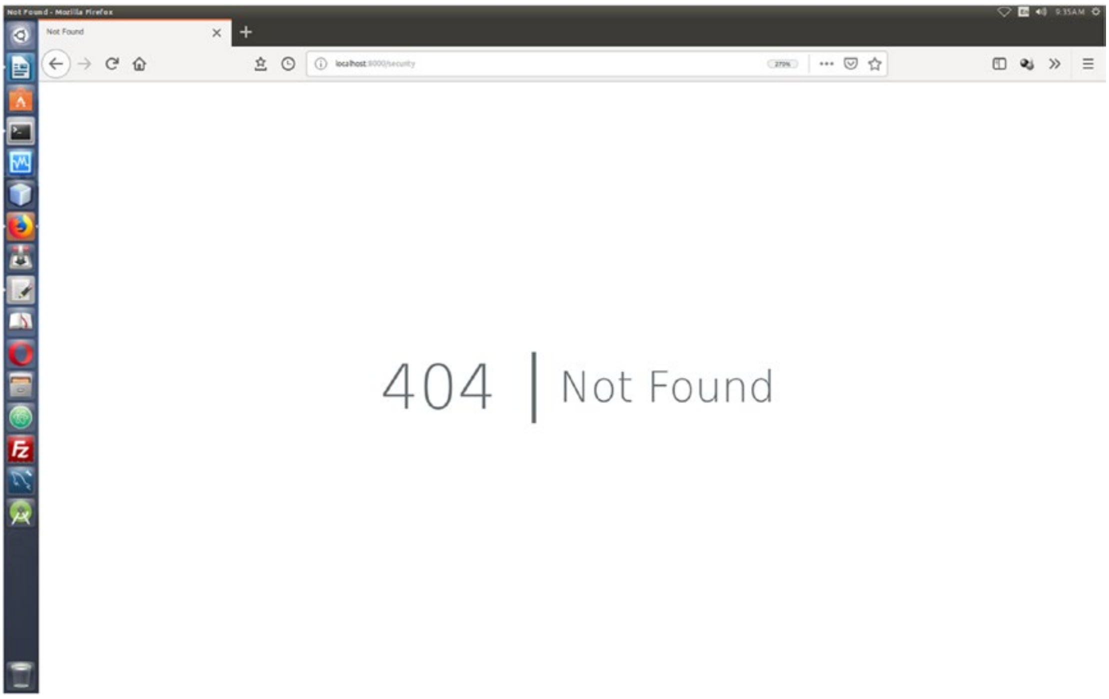


Figure A-8. *Error page in Laravel 5.8*

Improving Array and String Helper Functions

Array and string helper functions are going to be improved soon. The `array_*` and `str_*` global helpers have been deprecated already and will be removed in Laravel 5.9. There are `Arr::*` and `Str::*` facades that will be used instead (although there are packages that you will be able to use to maintain functionality if you don't want to change the existing code).

Changes in Caching

In the previous versions, caching was set in minutes, as shown here:

```
 // Laravel 5.7 - Store item for 10 minutes...
Cache::put('foo', 'bar', 10);
```

Laravel 5.8 uses seconds, so now you have to multiply 10 by 60 and change the previous code to this:

```
// Laravel 5.8 - Store item for 10 minutes...
Cache::put('foo', 'bar', 600);
```

There are some other minor improvements in Laravel 5.8. Please consult the documentation for the full list.

Where to Go from Here

You have learned about many important concepts in Laravel 5.8 in this book. I have used several different projects to explain Laravel concepts. Although you did not develop a complete project from beginning to end, you can find the full code of all the projects discussed in the code repository. Please download the code files for this book while you read and practice.

I have used the company/project/task management application to explain all the model relations along the authorization, authentication, and middleware.

Since Laravel has great support for the JavaScript framework Vue.js, I suggest you learn at least one JavaScript framework to facilitate more accomplishments in the future.

Learning a PHP framework like Laravel opens up many doors. I hope you will be able to use this knowledge base to move forward and develop some awesome applications.

Index