



Beginning Laravel

A beginner's guide to application
development with Laravel 5.3

—
Sanjib Sinha

Apress®

www.allitebooks.com

Beginning Laravel

A beginner's guide to application development with Laravel 5.3



Sanjib Sinha

Apress®

Beginning Laravel

Sanjib Sinha
Howrah, West Bengal, India

ISBN-13 (pbk): 978-1-4842-2537-0
DOI 10.1007/978-1-4842-2538-7

ISBN-13 (electronic): 978-1-4842-2538-7

Library of Congress Control Number: 2016962198

Copyright © 2017 by Sanjib Sinha

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director: Welmoed Spahr

Lead Editor: Nikhil Karkal

Technical Reviewer: Yogesh Sharma & Gaurav Yadav

Editorial Board: Steve Anglin, Pramila Balan, Laura Berendson, Aaron Black,
Louise Corrigan, Jonathan Gennick, Robert Hutchinson, Celestin Suresh John,
Nikhil Karkal, James Markham, Susan McDermott, Matthew Moodie, Natalie Pao,
Gwenan Spearing

Coordinating Editor: Prachi Mehta

Copy Editor: Brendan Frost

Compositor: SPi Global

Indexer: SPi Global

Artist: SPi Global

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.

Apress and friends of ED books 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 Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text are available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/. Readers can also access source code at SpringerLink in the Supplementary Material section for each chapter.

Printed on acid-free paper

IN MEMORY OF DR. BAIDYANATH HALDAR.

Sir, I truly miss you.

Contents at a Glance

About the Author	ix
About the Technical Reviewers	xi
Acknowledgments	xiii
■ Chapter 1: Composer	1
■ Chapter 2: Laravel Homestead, Virtual Box, and Vagrant	7
■ Chapter 3: File Structure	15
■ Chapter 4: Routing, a Static Method	21
■ Chapter 5: Controller Class	29
■ Chapter 6: View and Blade	41
■ Chapter 7: Environment	47
■ Chapter 8: Database Migration	49
■ Chapter 9: Eloquent	53
■ Chapter 10: Model, View, Controller Workflow	59
■ Chapter 11: SQLite Is a Breeze!	63
■ Chapter 12: Fiddly Feelings of Forms	65
■ Chapter 13: A CRUD Application	67
■ Chapter 14: Authentication and Authorization	81
■ Chapter 15: More About Validation	91
■ Chapter 16: Eloquent Relations	123

■ CONTENTS AT A GLANCE

- **Chapter 17: How Security and Authentication Work Together ... 153**
- **Chapter 18: How Request, Response Work in Laravel 5 161**
- **Chapter 19: Contracts vs. Facades 167**
- **Chapter 20: Middleware, Layer Filter, and Extra Security 173**
- Index..... 187**

Contents

About the Author	ix
About the Technical Reviewers	xi
Acknowledgments	xiii
■ Chapter 1: Composer	1
■ Chapter 2: Laravel Homestead, Virtual Box, and Vagrant.....	7
2.1 Installing Virtual Box and Vagrant	7
2.2 Installing Homestead Vagrant Box	9
2.3 Homestead Installation and Configuration	10
■ Chapter 3: File Structure	15
3.1 SOLID Design Principle.....	17
3.2 Interfaces and Method Injection	19
■ Chapter 4: Routing, a Static Method.....	21
4.1 Routing Best Practices	25
4.2 Named Routes.....	26
4.3 Organize Files Through Route	27
4.4 Advanced Concept of Routing and Anonymous Functions	28
■ Chapter 5: Controller Class.....	29
5.1 RESTful Controller	32
5.2 Role of a Controller	33
5.3 Resourceful Controller	34

■ CONTENTS

5.4	Controller, IoC Container, and Interface	35
5.5	Summary.....	38
■	Chapter 6: View and Blade	41
■	Chapter 7: Environment.....	47
■	Chapter 8: Database Migration.....	49
8.1	Summary.....	52
■	Chapter 9: Eloquent.....	53
■	Chapter 10: Model, View, Controller Workflow	59
10.1	Summary.....	61
10.1.1	Our Next Challenge.....	61
■	Chapter 11: SQLite Is a Breeze!	63
■	Chapter 12: Fiddly Feelings of Forms.....	65
■	Chapter 13: A CRUD Application	67
■	Chapter 14: Authentication and Authorization	81
■	Chapter 15: More About Validation.....	91
15.1	Conditional Rules	112
15.2	Custom Validation	113
15.3	How Form Validation Works	119
■	Chapter 16: Eloquent Relations	123
■	Chapter 17: How Security and Authentication Work Together ...	153
■	Chapter 18: How Request, Response Work in Laravel 5.....	161
■	Chapter 19: Contracts vs. Facades	167
■	Chapter 20: Middleware, Layer Filter, and Extra Security	173
	Index.....	187

About the Author

Sanjib Sinha writes stories and codes—not always in the same order.

He started with C# and .NET framework and won the Microsoft Community Contributor Award in 2011. Later, the open source software movement attracted him and he became a Linux, PHP, and Python enthusiast, specializing and working on White Hat Ethical Hacking.

As a beginner he always had to struggle a lot to find an easy way to learn coding. No one told him that coding is like writing—envisioning an image and bringing it down on the Earth with the help of words and symbols.

Through all his books, he has tried to help beginners from their perspective—as a beginner.

About the Technical Reviewers

Yogesh Sharma I am a web developer, IT consultant and an entrepreneur based in Pune, India. I have experimented with many IT paradigm liked Cloud Services, NoSQL, Middleware but programming is at my heart. Graduated from Vidyalankar School of Information Technology, I am currently employed with a fortune 500 company as Senior Infrastructure Engineer.

I really like to see the big picture with eye for distinctive intricacies. I have 8 years of cumulative experience across various domains in IT and also served clientele across the globe developing projects ranging from Asset Management to Manufacturing and Logistics. I like to tinker with IoT, AI, Speech and Simulation. If I am not these, you might find me brewing an espresso shot!"

Gaurav Yadav is a Full Stack Web Developer and blogger. Sportsperson by heart and loves football. He has experience with various frameworks in php, python and javascript. Loves to explore new frameworks and evolve with the trending technology.

Acknowledgments

KARTICK PAUL, SYSTEM MANAGER, AAJKAAL, KOLKATA: Without his persistent and inspiring help, I could not have written this book.

CHAPTER 1



Composer

Composer is a dependency management tool in PHP (Figure 1-1). For any PHP project you need to use your library of codes. Composer easily manages that task on your behalf, helping you to declare those codes. You can also install or update any code in your library through Composer. Please visit <https://getcomposer.org> for more details.

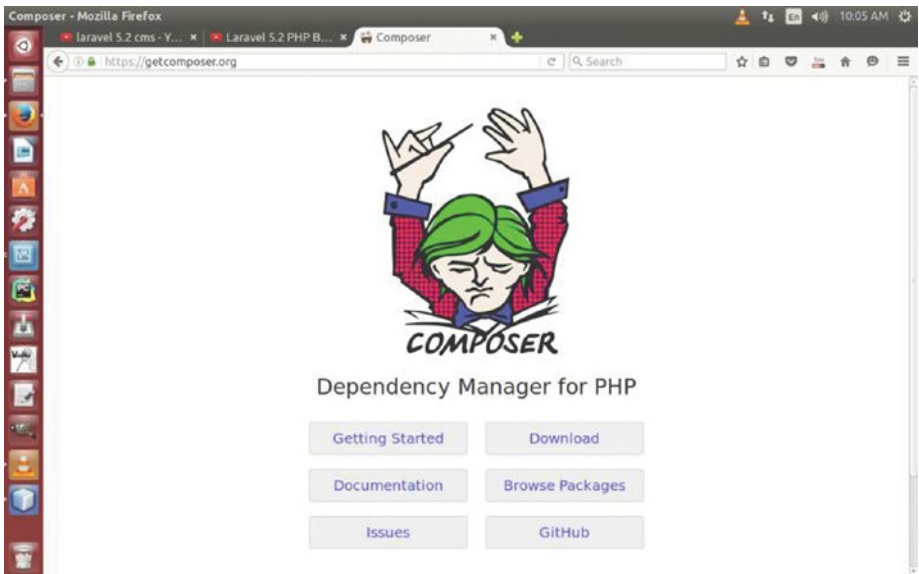


Figure 1-1. Composer home page

In the opening page of <https://getcomposer.org>, click the 'getting started' link (Figure 1-2).

Electronic supplementary material The online version of this chapter (doi:[10.1007/978-1-4842-2538-7_1](https://doi.org/10.1007/978-1-4842-2538-7_1)) contains supplementary material, which is available to authorized users.

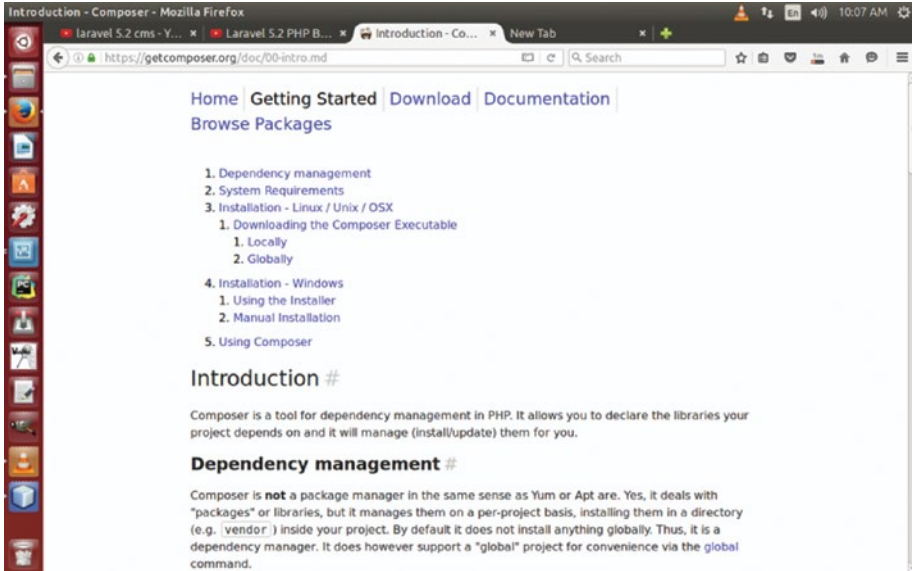


Figure 1-2. Getting started page in Composer website

In the page shown in Figure 1-2, you find two links: ‘locally’ and ‘globally’. It stands for two options. Suppose you don’t want to run Composer globally or centrally in your system. In that case, you have to download and install Composer each time for every project. But the global option is always preferable because once Composer is installed in your system bin folder, you can call it anytime for any project.

If you are already accustomed to any Linux distribution like Ubuntu, you know that for any local PHP project we used to go to ‘/var/www/html’ folder. Suppose we are going to build a simple Laravel project and we want to name it ‘MyFirstLaravelProject’. Open up your Ubuntu terminal (ctrl+alt+t) and go to that folder first.

To reach there, you need to type the following command on your terminal:

```
cd /var/www/html/
```

Once you’ve reached it, you can make a directory here with a simple command:

```
sudo mkdir MyFirstLaravelProject
```

It will ask for your ‘root’ user password. Type the password and a folder called ‘MyFirstLaravelProject’ will be created.

Next in this folder we’ll download and install ‘composer’. Considering you are a beginner, for the sake of brevity I want to download and install Composer locally on our Laravel project.

Next, issue these two commands, one after another. First you type this:

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

```

It'll take some time. Next type this:

```
sudo php composer-setup.php

```

It'll organize your Composer setup file to go further. Actually your Composer is ready to download packages for your coming project. You can test it by creating a 'composer.json' file inside your 'MyFirstLaravelProject' folder. In that 'composer.json' file type this:

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

What does this mean? It means you're installing 'monolog' PHP package for your Laravel project. Will it come to any immediate use? The answer is 'NO'. We're actually testing our Composer installer and want to see how it works.

Now you can issue the command that will install 'monolog' package for you. Type this command on your terminal:

```
sudo php composer.phar install

```

It'll take a little time to install the 'monolog' package. It depends on your Internet speed.

After the installation is over you'll find a 'vendor' folder and a few 'composer' files inside your project. Feel free to discover what is inside the 'vendor' folder. There you'll find two folders: 'composer' and 'monolog'. Again you can see what they have inside them. As a beginner it's an endless journey to discover new things. Try to get acquainted with everything new you have found.

The time has come to install Laravel 5.2 through Composer. You can install Laravel just like monolog. It means that you can write that instruction in your 'composer.json' file and just update your Composer. But as a beginner I recommend following the simplest method.

Open up your terminal and write the following:

```
sudo composer create-project --prefer-dist laravel/laravel blog

```

It'll install Laravel latest version in the folder 'blog' in your Laravel project 'MyFirstLaravelProject'. Once it's done you'll get this message on your terminal:

```
Application key
[base64:FrbQTSPezY8wQq+2+bZ/ieA8InA4KjA9N4A44AMbqas=] set
successfully.
```

It's a random key generated each time you install Laravel. It means you have successfully installed Laravel 5.3.

- First step completed: you've installed Laravel in the `./var/www/html/MyFirstLaravelProject/blog` folder. Now you can go inside that folder and issue a Linux command `ls -la` to see what's inside. You can also type `php artisan serve` command to run your first Laravel application so that you can go to <http://localhost:8000> to see the welcome page. This installation has been done locally.

There is another, easier method. You can install Composer globally in your Linux system. Open your terminal and make a directory named 'Code' on the desktop. Open up your terminal and type this:

```
cd /Desktop
```

Now you are inside your desktop. You're going to make the 'Code' directory there. Type the following:

```
mkdir Code
```

Now you must go inside your 'Code' directory by writing `cd Code` on your terminal. Next, inside 'Code' folder make a directory 'test', where you'll create your first Laravel application.

Not only that, after the 'Laravel/Homestead' is installed, you can type <http://test.app> to run your first Laravel application. I'll show you that step by step.

Now it's time to install Composer globally. Type these commands one after another:

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

```
sudo php composer-setup.php
```

```
sudo php composer.phar install
```

It'll take a few minutes to install Composer globally. Once you have done it, you can create any Laravel project anywhere.

Next you can create your first Laravel project inside the 'Code/test' folder by typing this command:

```
sudo composer create-project --prefer-dist laravel/laravel blog
```

Inside the 'test' folder, the Laravel project is installed as 'blog'. While installing, you can change this name to your choice.

It'll take a few minutes depending on the speed of your Internet connection.

Once done, it'll give this message:

```
Application key  
[base64:FrbQTSpezY8wQq+2+bZ/ieA8InA4KjA9N4A44AMbqas=] set  
successfully.
```

You have installed the latest version of Laravel 5.2.4 on your `~/Desktop/Cd/test/blog` folder. And you have done it globally. Feel free to create any folder and subfolders anywhere in your machine and install Laravel through Composer.

- Second step completed: Laravel installed in `~/Desktop/Code/test/blog` folder, and you've done it globally.

In the next chapter, we'll learn a little about installing Laravel Homestead.

CHAPTER 2



Laravel Homestead, Virtual Box, and Vagrant

Laravel Homestead is an official, prepackaged Vagrant box. An absolute beginner may find that concept a little bit quirky. You can imagine it as a scaffold platform or magical box that contains everything for building Laravel applications on your local machine. I encourage you to search and learn about the Laravel Homestead package. If you have Laravel Homestead installed, you need not worry about the latest PHP and Linux versions. It also comes with a web server and all types of server software that you need to develop some awesome PHP applications besides Laravel!

Before running Laravel/Homestead you must have Virtual Box 5.x:

<https://www.virtualbox.org/wiki/Downloads>.

You also must install Vagrant:

<http://www.vagrantup.com/downloads.html>.

2.1 Installing Virtual Box and Vagrant

The first question that comes to our mind is the following: why do we need a “virtual box” when we have a default operating system in place? There are several reasons. The most important reason is that in Virtual Box we can play with any operating system without any fear of messing it up, even breaking it. There is every possibility that while testing a hacking tool we could break a system. I encourage you to do that. It is a virtual machine. So, go ahead. Test everything that comes to mind. Another great reason of using Virtual Box is the safety. When you visit a web site you might consider it to be safe but in reality it could not be so. But nothing matters in the case of Virtual Box. It is not your original machine with confidential data. Visiting unsafe web sites is not annoying (or worse) any more.

There is only one thing you need to remember. Stay within the law. While testing your hacking tools or running codes, you can't jeopardize any other system.

The Oracle Virtual Box official web site offers plenty of download options. You can choose any one of them. According to your OS, you go to the “download” section and see what is available for you. From Figure 2-1, you will gain an idea of how to proceed further.



Figure 2-1. Virtual Box download section for Linux hosts

The selected line in Figure 2-1 shows the default operating system I am running currently. That is “Ubuntu 14.04 (Trusty)” and the architecture is “AMD64”.

Virtual Box is very easy to install. Whatever your OS is—Mac OS X, Windows, or Linux—you can install it. First you need to know about your operating system itself. It could be either 32-bit or 64-bit architecture. In any Linux distribution it is extremely easy to learn. Just open up the terminal and type “uname -a”.

The terminal will spit out some vital information that includes all data regarding the current default system. In this case, Linux is version 3.19.0 and the superuser’s name is “hagudu”; finally, it also indicates what type of system architecture is being used.

As in my case, it is “x86_64” which stands for 64 bit. In the Virtual Box official download page for all Linux distribution, you first download the required packages and then install them according to the nature of your OS. For Red Hat, Fedora, or any Linux distribution belonging to that category, you will notice that the last extension is “.rpm”. In that case you can move to the Virtual Box folder and issue commands like “rpm -i” or “yum install” in case you run Red Hat or Fedora.

But there are simpler methods to install Virtual Box.

For absolute beginners, it is very helpful to run “UBUNTU” Linux distribution as your default OS. You can install Virtual Box from the software center directly without opening up the terminal or issuing any command.

“UBUNTU” software center has many categories. One of them shows the “Installed” software.

You may not find it there by default. In that case it is extremely easy to install. You can just type “Virtual Box” on the search text box and it will pop up. Move ahead and press the installation button and it will get installed in your system. Installing Vagrant is also easy. Go to the official web site and download it according to your operating systems. The installation process is also easy. For Ubuntu, just extract the content anywhere and install it according to the procedure mentioned in the site. Search the Internet you’ll get a ton of guides.

Through Virtual Box you can run different operating systems on your machine (Figure 2-2).



Figure 2-2. A Virtual Box running Kali Linux and Windows XP

Having installed Virtual Box you can issue 'vagrant -v' command on your terminal and the message will pop up as 'vagrant 1.8.5'. It's running.

Now it's time to install Laravel/Homestead.

2.2 Installing Homestead Vagrant Box

Once you have installed Virtual Box and Vagrant, you can add 'laravel/homestead' box to your Vagrant box using this command on your terminal:

```
vagrant box add laravel/homestead
```

It will take some time, anywhere from 15 minutes to an hour, depending on your Internet speed.

2.3 Homestead Installation and Configuration

Next you can install Homestead by cloning Homestead repository to your '/home/Homestead' folder using this command:

```
git clone https://github.com/laravel/homestead.git
Homestead
```

It'll take a few seconds. Next you need to initialize your Homestead and create the configuration file.

```
bash init.sh
```

Next, run an 'ls -la' command to find out the hidden './homestead' directory. Type 'cd ./homestead' command to enter into it and run 'ls -la' command again. You'll find a file called 'Homestead.yaml'. You may consider this file as the brain of your 'laravel/homestead' local developmental environment.

Through this file, you can instruct the local web server. You can mention the path of your project root. You can decide the name of your local application. I think it's always wise to adopt the same name that you are going to use in your production environment. Suppose your final application in the production level will be named 'www.example.com'; in that case, it's good to use the same name locally so that you can type <http://example.com> in your browser to test the application locally.

Before editing 'Homestead.yaml' file, you can do two more things. First, check your 'laravel/homestead' version and if necessary run this command: 'sudo composer global require laravel/homestead:v2.0.8'. Always try to keep the latest one. Check it in the Internet. Next, you can run this command also: 'export PATH=~/.composer/vendor/bin:\$PATH'. It'll help you to run 'homestead' command from anywhere on your system.

Now you need to edit the 'Homestead.yaml' file for keeping two major things in place. The first is the provider. Run 'sudo gedit Homestead.yaml' command to open up the file on the text editor. By default you'd see 'provider: virtualbox' in your file. So you need not change this. Next, you check this part.

The second part is very important. By default the 'Homestead.yaml' file comes up with this folder and site structure:

```
folders:
```

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

```
sites:
```

- map: homestead.app
- to: /home/vagrant/Code/Laravel/public

We have already installed Laravel at the 'Code/test/blog' folder on the desktop. So we need to add that in this folder and site section first.

folders:

- map: ~/Code
to: /home/vagrant/Code
- map: ~/Desktop/Code/test/blog
to: /home/vagrant/ Desktop/Code/test/blog

sites:

- map: homestead.app
to: /home/vagrant/Code/Laravel/public
- map: test.app
to: /home/vagrant/ Desktop/Code/test/blog/public

The added lines are marked in red. Please note that I have mentioned the full path. Wherever you keep your Laravel application, you need to mention the full path. So, we have almost come to the close.

You've probably noticed that we've named our application 'test.app'. Next you need to add the "domains" for your local sites to the 'hosts' file on your machine. The 'hosts' file will redirect requests for your Homestead sites into your Homestead machine. On Mac and Linux, this file is located at '/etc/hosts'. Open this file in your text editor.

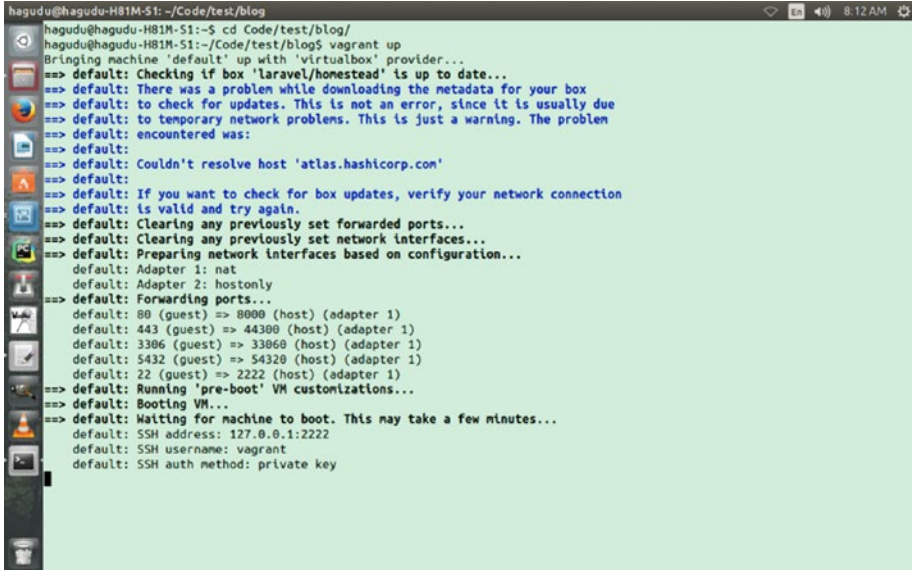
```
sudo gedit /etc/hosts/
```

Generally it comes up with two lines on the top. You must add your 'test.app' after the two lines.

```
127.0.0.1 localhost      127.0.0.1      hagudu-H81M-S1    192.168.10.10
test.app
```

Everything done, you may now fire up Vagrant and run your site. Go to the Laravel folder 'cd /Desktop/Code/test/blog' and issue this command: 'vagrant up'.

The terminal usually looks like the image in Figure 2-3. It may look different depending on your operating system.



```

hagudu@hagudu-H81M-S1: ~/Code/test/blog
hagudu@hagudu-H81M-S1:~$ cd Code/test/blog/
hagudu@hagudu-H81M-S1:~/Code/test/blog$ vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Checking if box 'laravel/homestead' is up to date...
==> default: There was a problem while downloading the metadata for your box
==> default: to check for updates. This is not an error, since it is usually due
==> default: to temporary network problems. This is just a warning. The problem
==> default: encountered was:
==> default:
==> default: Couldn't resolve host 'atlas.hashicorp.com'
==> default:
==> default: If you want to check for box updates, verify your network connection
==> default: is valid and try again.
==> 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: Adapter 2: hostonly
==> default: Forwarding ports...
default: 80 (guest) => 8000 (host) (adapter 1)
default: 443 (guest) => 44300 (host) (adapter 1)
default: 3306 (guest) => 33060 (host) (adapter 1)
default: 5432 (guest) => 54320 (host) (adapter 1)
default: 22 (guest) => 2222 (host) (adapter 1)
==> default: Running 'pre-boot' VM customizations...
==> 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

```

Figure 2-3. The terminal after issuing ‘vagrant up’ command

It normally takes a few seconds to fire it up. Give it that time, and next you can safely type <http://test.app> on your browser to see the Laravel welcome page.

Being on your Laravel folder, you may run ‘php artisan serve’ command to run the same application. In that case, you must type <http://localhost:8000> on your browser.

However, there are lots of differences with the Homestead server. When you run Homestead you get the latest php version that is php 7.

Look at the following image. I have simply kept the ‘phpinfo()’ method in my ‘test.app’ Laravel home page (Figure 2-4).

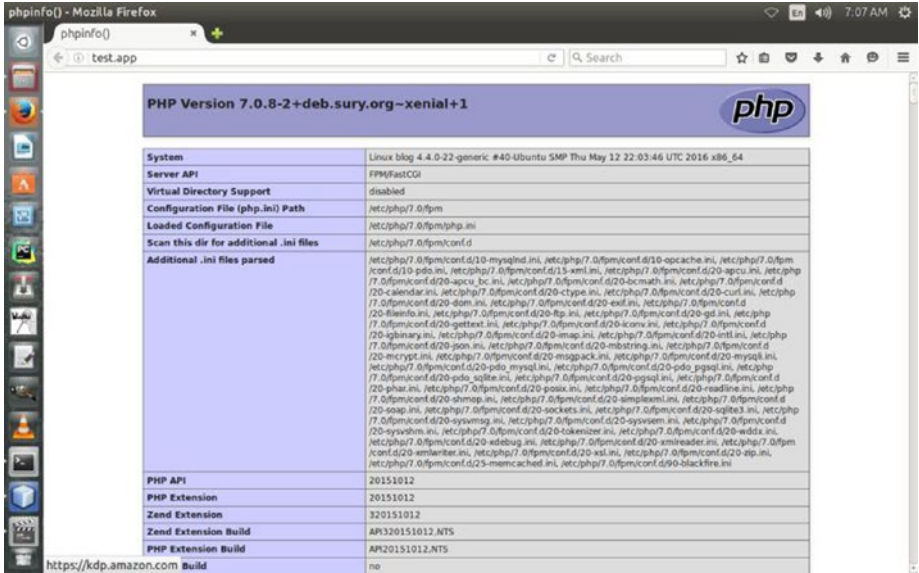


Figure 2-4. `test.app` running the `php` 7 information

I hope you can do it. If by any chance if you're stuck, feel free to drop me a line at sanjib12sinha@gmail.com. I'll definitely try to help.

CHAPTER 3



File Structure

Let us see what's inside the installed Laravel folder 'blog'. It has folders like 'app', 'bootstrap', 'config', 'database', 'public', 'resources', 'storage', 'tests', 'vendor', and a few more files, including a 'composer.json' file.

Let us first see how the file structure looks (Figure 3-1).

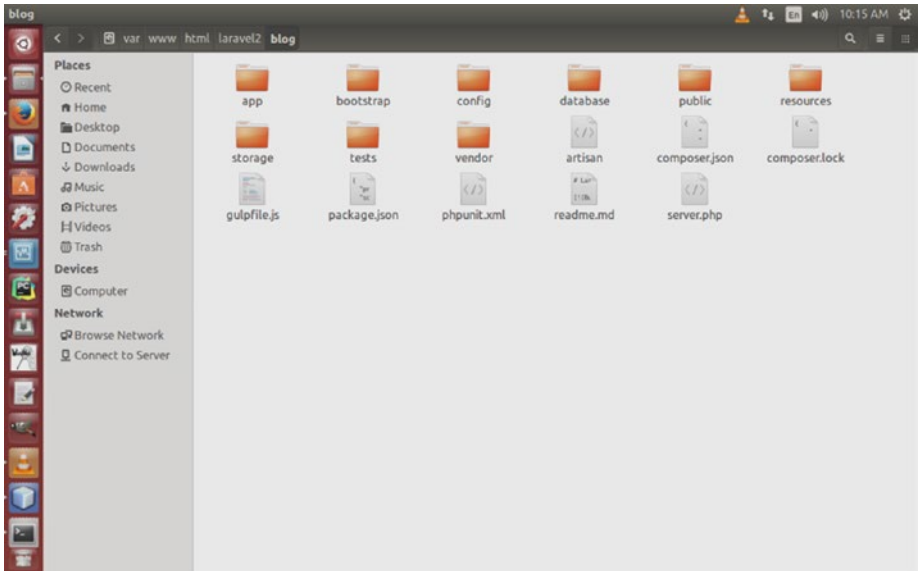


Figure 3-1. *Laravel 5.3.18 file structure*

As you see, each folder and its included files have their own roles clearly defined. You should not try to change or tweak any source code.

The 'app' folder is extremely important. It has, again, many significant folders inside. Presently we'll look forward for the 'Http' folder that has controllers, middleware, models, and the 'routes.php' file. Actually, we put our business logic in this folder through route and controllers. It helps us maintain the loosely coupled object-oriented design pattern.

The 'bootstrap' folder is needed for the startup of Laravel, and you need 'config' folder for many configuration files. As a PHP programmer, you know that session configuration or authentication configuration is important. Laravel manages it through this 'config' folder.

The role of 'database' folder is also very vital for migrations and seeds, which we will discuss later in great detail. Presently you may think of database migrations as PHP files that write SQL codes and perform database operations directly on your application. You need not go to any MySQL interface like PHPMYADMIN. You write PHP codes and through the Laravel command line you can execute database operations. You can fill database tables with data, manipulating them accordingly. I think it's one of the greatest features of Laravel.

In the 'public' folder we have files that are available publically: '.htaccess', 'robots.txt', 'favicon.ico', and 'index.php'. These files play important roles in keeping your project in the search engines. The 'resources' folder has the important sub-folders like 'views', where your viewable PHP/HTML codes are stored. In 'storage,' cache and log files are kept, and the 'tests' folder is exclusively for unit testing. Finally, you meet the 'vendor' folder again here, but it's only for the third-party packages that Laravel uses.

I encourage you to enter each folder, open every file, and see how they are written. You only remember one thing: never change any source code.

Our Laravel installation is complete. We've first installed it in the '/var/www/html/MyFirstLaravelProject/blog' folder. Next we have installed it in '/Desktop/Code/test/blog' folder. The first one will be used for the MySQL database operations and the second one will be used for SQLite database operations.

All we now need to see is that it works perfectly. Let us go the first one. Open up your terminal and write the following:

```
cd /var/www/html/MyFirstLaravelProject/blog
```

We have reached our Laravel project. Now issue this command:

```
php artisan serve
```

It will start the local development environment server so that if you open your web browser and type <http://localhost:8000>, it will open up the home page of your Laravel project.

It should look like this (Figure 3-2):



Figure 3-2. *Laravel 5.3.18 home page*

All browsers usually give simple HTML output. In that sense, there must be some HTML codes hidden somewhere in our Laravel project.

In Chapter 4, we'll see how we can change this HTML output. Besides that, we'll very briefly discuss some points of 'views' folder where we actually need to store these HTML codes.

Later in the book, in the 'views and blades' chapter, we'll learn it in detail.

3.1 SOLID Design Principle

Yes, this is quite an advanced thing that you need to learn better beforehand. From Laravel 4 the SOLID design principle has been maintained, and in Laravel 5 it becomes more familiar so that you can avoid hard coding and write cleaner codes. Let us see what this SOLID design principle is.

This book is not the place to put forward a detailed description of the SOLID principle. But at least we can present something about it in a nutshell.

SOLID consists of the five design principles articulated by Robert "Uncle Bob" Martin. Here they are in brief, one by one. In the final part I will discuss it in detail, and hopefully by that time you will have become acquainted with the basic principles of Laravel application logic.

SOLID stands for

- 1) Single Responsibility Principle
- 2) Open Closed Principle
- 3) Liskov Substitution Principle
- 4) Interface Segregation Principle
- 5) Dependency Inversion Principle

The Single Responsibility Principle means a class should have one, and I mean only one, reason to change. Limiting class knowledge is important. The class's scope should be narrowly focused. A class would do its job and not be affected at all by the change that takes place on its dependencies. Remember, if we can build a library of small classes with well-defined responsibilities, our code will be more decoupled and easy to test and run. The Open Closed Principle means a class is always open for extension but closed for modification. How is that? Nothing except that any changes to behavior should be made without modification of source codes. If you can do your job without touching the source code, then you are following the Open Closed Principle! Remember what Uncle Bob says: "Separate extensible behavior behind an interface and flip the dependencies." The thing is that any time you modify your code, there is a possibility to break the old functionalities completely, adding new bugs.

But if you can plan your application in the beginning based on Open Closed Principle, you could modify your code base as quickly as possible without getting affected. What is the Liskov Substitution Principle? Don't get frightened. This looks intimidating, but as a principle it is extremely helpful and easy to understand. It says: Derived classes must be substitutable for their based class. It means objects should be replaceable with instances of their subtypes without altering the correctness of program. If you can't follow that, just move on; I will explain these principles in detail with examples and screenshots, so that the picture will be much clearer.

The Interface Segregation Principle is an echo of Singular Responsibilities. If it is broken, Singular Responsibility is broken. In a nutshell, it says that interface is granular and focused. No implementation of interface should be forced on methods that it does not use. Accordingly, break into small interfaces as you require them for your implementation. Plan it before and enjoy the decoupled easy-going ride.

Finally, the Dependency Inversion Principle states that highlevel codes should not depend on lowlevel codes. Instead the highlevel code depends on 'Abstraction' that acts as a middle-man between highlevel and lowlevel. The second aspect is that abstraction does not depend upon details but details depend upon abstractions. For beginners, these principles may look not comfortable enough, but don't worry, as we will discuss it in detail with examples; by that time you will have gained enough confidence to tackle this conceptual lingo. You have already found the terms "Interface" and "Abstraction" more than once, perhaps many times, to be a deserving candidate for discussion. So let's spend some time with those omnipresent terms that we so often come across in our Laravel application.

3.2 Interfaces and Method Injection

Abstraction in OOP involves extraction of relevant details. Consider the role of a car salesman. There are many types of consumer. Everyone wants to buy a car, no doubt, but each one has different 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 salesman knows all the details of the car but does he repeat the list one by one until someone finds his or her choice? No. He presents only the relevant information to the potential customer. As a result the salesman practices “Abstraction” and presents only relevant details to customer.

Now consider abstraction from the perspective of a programmer who wants a user to add items to list. Abstraction does not mean that information is unavailable but it assures that the relevant information is provided to the user.

PHP 5 introduces abstract classes and methods. Classes defined as abstract may not be instantiated and any class that contains at least one abstract method must also be abstract. Remember that abstract methods can not define the implementation. On the other hand, object interfaces allow you to create code which specifies which methods a class must implement, without having to define how these methods are handled.

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. I have a .NET background and found the usage of interfaces there almost ubiquitous. And in Laravel, the injection of interfaces to the classes is seen frequently, so you better get acquainted with the conceptual background.

In Laravel Interface is considered as a Contract.

Contract between whom? And why? Because an interface does not contain any code but it only defines a set of methods that an object implements. Having said that, I hope you now understand that they are interrelated. We talked about maintaining a library of small classes with clearly defined scopes, and this is achievable with the help of Interfaces. As the book progresses, you will find a lot of examples of Interfaces, so don't worry, you will find these two words pretty often in the examples. Till then, have patience and read on.

For a clear picture, I would like to give a small example so that we can understand this property of encapsulated contract behavior of Interface quickly.

Let us imagine I have a 'Connection' folder inside a folder called 'Bengaliana' in which I have a connection class that would connect us to the database 'sanjib' and retrieve some data from the respective tables 'users' and 'tasks'.

I don't want to make the 'ConnectionClass' know our datatarget. All it will do is just get the connection and retrieve the one attribute from a table. The attribute and table name I would like to supply dynamically so that from one method I can retrieve many kinds of data. It could be user names or simple task titles, et cetera. That will also conform to the homomorphism nature of our objects.

In Laravel 5, as document says, all major components implement interfaces which are located in the 'illuminate/contracts' repository. This repository has no external dependencies. Having a centralized set of interfaces make you free to use alternative optional decoupled classes and do some dependency injection more freely without Laravel Facades. It sets your choices more open and user friendly. Well, there are a lot of new staffs you will find as you progress and the new features will make your journey absolutely enjoyable. To name a few, there are Route Cache, Middleware, Controller Method Injection, and many more.

In Laravel 5, authenticating users becomes easier, and the user registration, authentication, and password reset controllers are now included out of the box so that they can be easily used. Okay, enough introduction. Now it is time to catch the web artisans and do some codes so that we can make some awesome applications in the future.

CHAPTER 4



Routing, a Static Method

Routing is the concept of setting up a new URI like <http://localhost:8000/hello>. It will take you to a destination web page. Laravel makes it extremely simple. It's a static method that accepts two things: a URI and an anonymous function or Closure.

In the 'app/Http/routes.php' it's been defined primarily. Start your favorite text editor and open the file. What you see?

You see some code like this:

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

Here 'Route' is a class that has a static method 'get' that returns a 'view' method which presents a web page. When a user visits the home page he is taken to the 'welcome' page. This 'welcome' is actually a PHP file: 'welcome.blade.php'. It was stored by default in the 'views' folder while we had been installing Laravel. When we discuss the concept of 'view' and 'blade' template engine you'll fully understand what's happening in the presentation layer.

Using HTTP protocol you can send another request instead of the default route.

```
Route::get('/', function () {
    return 'welcome';
});
```

It will simply return the word 'welcome' on your home page. Try it or anything else.

Look at the code; there is an anonymous function also. As a PHP programmer I assume you know about anonymous function or closure. It's a very useful function we often use in building applications. Laravel also uses this concept.

Now we have not yet defined our route, so if we hit the browser on the URL <http://localhost:8000/hello> we will get an error page. So let us create it first. Go to the 'routes.php' file and add this code:

```
//requesting a new page
Route::get('/hello', function () {
    echo '<h1>Hello</h1>';
});
```

Let us see how it looks and if we can add any new HTML code into it to change the look.

Primarily it looks like this (see Figure 4-1):

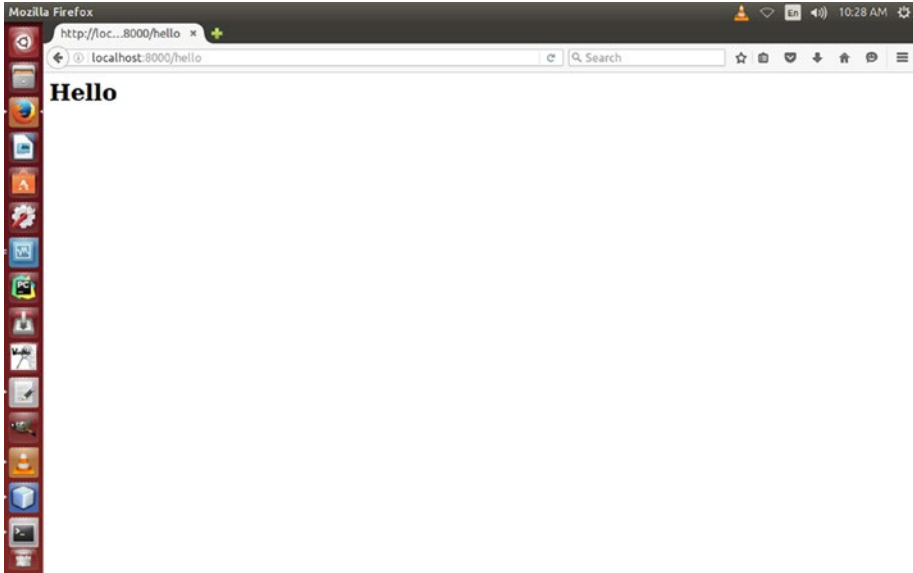


Figure 4-1. Routing to <http://localhost:8000/hello> URL

Now we'd like to add more CSS style into it. I change the previous code into this:

```
//requesting a new page
Route::get('/hello', function () {
    echo "<html>"
    "<head><title>Laravel</title>"
    "<style>"
        "html, body {"
            "height: 100%;"
        "}"
    "body {"
        "margin: 0;"
        "padding: 0;"
        "width: 100%;"
        "display: table;"
    "}"
    "</style>"
    "</head>"
    "<body>"
    "</body>"
    "</html>"
})
```

```

        font-weight: 100;
        font-size: 20px;
        font-family: 'Lato';
    }
    .container {
        text-align: center;
        display: table-cell;
        vertical-align: middle;
    }
    .content {
        text-align: center;
        display: inline-block;
    }
    .title {
        font-size: 110px;
    }
</style>
</head>
<body>
    <div class='container'>
        <div class='content'>
            <div class='title'>Hello, Dear Reader</div>
            <div class='container'>
                <div class='content'>How are you? I hope you would
                    love this book!
                </div>
            </div>
        </div>
    </div>
</body>"
"</html>";
});

```

This code is quite big. It's because I have added a small CSS style into it. You need not write this code. Just try to understand the power of this small file 'routes.php'. You can virtually add any functionality into your application through this file. But never do this. The file 'routes.php' is not meant for this. Laravel has made separate places to put your CSS styles and HTML pages. We'll see to it later in detail. Just before that we have just wanted to play with it.

Now it looks completely different in your browser. Just have a look (see Figure 4-2):

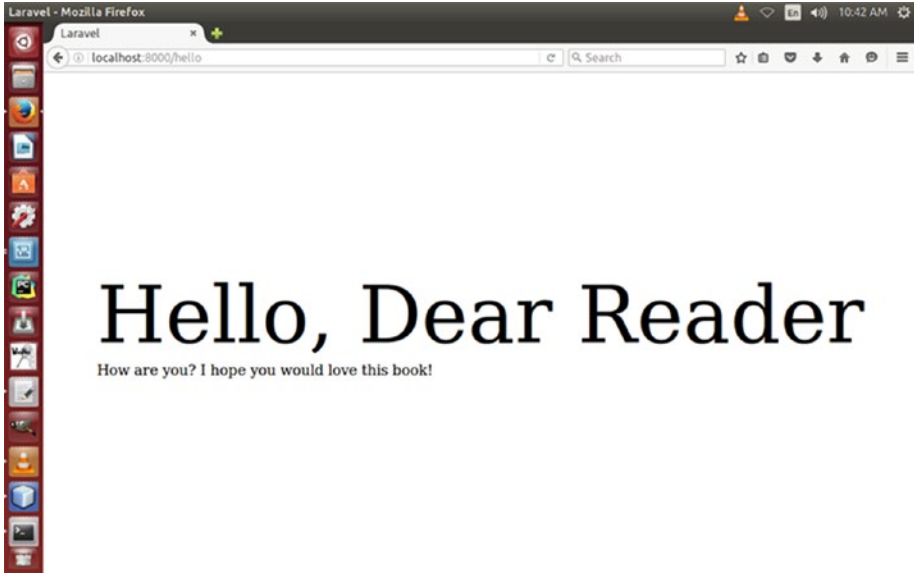


Figure 4-2. Routing to <http://localhost:8000/hello> URL

With a new CSS style it looks very special. Later we will give the same output using Laravel’s ‘blade’ template engine.

Now, routing has many other important functions. Let us consider them one by one.

First of all, we can add route parameter and pass data through the URL. It’s something like you have an <http://localhost:8000/hello> page and want to say ‘hello’ to your friend. Laravel manages it nicely. Through the anonymous function we can pass this parameter such as <http://localhost:8000/hello/reader> and the output will be ‘Hello reader.’ Let us see how we can do that.

Add this code to your ‘routes.php’:

```
Route::get( 'hello/{name}', function ($name){
    echo "Hello " . $name;
});
```

Now you can pass any data through the URL <http://localhost:8000/hello/reader> as route parameter. I passed ‘reader,’ you can pass any other data as you like.

I hope you get a basic idea how to use Route class. You also have come to know that Route class has some static methods like ‘GET’ and other methods. Through these methods Route class help the user reach her URL destination.

Other routing methods are like this:

```
Route::get();
Route::post();
Route::patch();
Route::put();
```

```
Route::delete();
Route::any();
```

I find the 'any()' method very interesting. You can register any route that responds to any HTTP verbs with this method.

```
Route::any('any', function (){
    return "Anything is possible if you try hard!";
});
```

Go to <http://localhost:8000/any> and see this simple output.

4.1 Routing Best Practices

Suppose you want to protect your application throughout from cross-site request forgery (better known as CSRF). What you can do? Is there any single command we can issue to protect our site from the 'Crackers' or 'Bad Guys'? Yes, there are. Let us see how we can do this. Write this code in your routes.php:

```
Route::when('*', 'csrf', ['post', 'put', 'patch']);
```

Voila! This is all you need to protect your site and from the next time you launch your Laravel application, it will take care of all types of CSRF.

Especially in a large application, where lots of users put plenty of posts and your server is busy, this is a major headache. But wait a minute. There are lot of uncommon things in 'Route' class; let me call it magical, waiting for us to explore it.

First of all, we will discuss about 'implicit' and 'explicit' routing. If you have a 'CodeIgniter' experience, you probably have seen instances of 'implicit' routing. When we write like this:

```
Route::controller('admin', 'AdminController');
```

It implicitly redirects us to 'View/admin' folder through 'AdminController'. Okay, you are new to the concept of Controllers, but don't worry; in the next chapter we will tackle all the controller staff, so read on. I suggest that you reread this chapter later to understand the magic better.

The question is as follows: can we explicitly 'route' this? Yes, we can. Let us try:

```
Route::get('test/test', 'HomeController@showTest');
Route::get('home/index', 'HomeController@showIndex');
Route::get('home/about', 'HomeController@About');
Route::get('home/contact', 'HomeController@showContact');
Route::get('php/phptraininginkolkata', 'PhpController@showIndex');
Route::get('php/variableanddatatype', 'PhpController@phpFirstPage');
Route::get('codeigniter/codeignitertrainingin kolkata',
'CodeIgniterController@showIndex');
```

```
Route::get('codeigniter/howtostartcodeigniter', 'CodeIgniterController@
CIFirstPage');
Route::get('wordpress/howtostartwordpress', 'WpController@showIndex');
```

Let me pick up the last line:

```
Route::get('wordpress/howtostartwordpress', 'WpController@showIndex');
```

It says: in the 'View/wordpress' folder, we have a page like 'how-to-start-wordpress.blade.php' and our controller 'WpController' got a public function called 'showIndex', which has something inside it so that our end point is 'howtostartwordpress.blade.php'.

Now the question is as follows: can we group this long list of controller routing?

Yes, we have a handy tool called `Route::group()`. Suppose we are only concerned with the 'home' controllers. We can group it; no problem. The entire 'home' blade can come under one group like this:

```
Route::group(['prefix']=>'home', function()
{
Route::get('/', 'HomeController@showIndex');
Route::get('/about', 'HomeController@About');
Route::get('/contact',
'HomeController@showContact');
});
```

You see, inside the group, we need not write 'home' anymore. It is explicitly told to our router. In the command line you can always check how your work is proceeding through this command:

```
php artisan routes
```

Ultimately, it shows all the controllers I have used in my 'routes.php' file and written before. To get hold of all 'home' controller we can do another thing. Considering that 'home' as my 'resource' I can write like that:

```
Route::resource('home', 'HomeController');
```

There are more to come. Another best practice is 'naming a route'.

4.2 Named Routes

Suppose you have a Login page and you are going to use session for accomplishing the task. Now to start with, we can write like this to reach our Login page:

```
//example of named routes
Route::get('/', function(){
return link_to_route('session/create', 'Login');
});
```

We can write the same route using our controllers like this:

```
Route::get('session/create', 'SessionController@create');
```

But all these are normal procedures that we have seen before. Then, you may have asked, where is the named route?

Okay, here it is:

```
Route::get('session/create', ['as'=>'create',
    'use'=>'SessionController@create']);
//equivalent to
Route::get('/', function(){
    return route('create');
});
```

And there are a lot of staff still waiting for you. It is your preference what you would have used, as the choice is yours.

```
Route::get('register', [
    'before'=>['guest'],
    'as'=>['register'],
    'use'=>['SessionController@register']
]);
```

You can write it like this also:

```
Route::get('register', [
    'as'=>['register'],
    'use'=>['SessionController@register']
])->before('guest');
```

4.3 Organize Files Through Route

Yes, you can organize your files through your Route. And sometimes that could be very handy. Suppose you have a folder structure like this:

```
app/routes/admin.php
app/routes/login.php
app/routes/register.php
```

Now we can declare this structure in your 'routes.php' in a single command like this:

```
(File::allFiles(_DIR_, '/routes') as $partial)
{
    require_once $partial->getPathname();
}
```

4.4 Advanced Concept of Routing and Anonymous Functions

Being able to define your own function by yourself is a great thing that you can do as a programmer, and you have to do it in the course of developing an application. In Laravel we have seen the concept of closures. What is that? Seasoned programmers know it very well but for the beginners it requires some more explanation. This is nothing but called an anonymous function. It is also called 'lambdas'. An anonymous function is nothing but a function without a name.

```
$hello = function($name){
echo "Hello " . $name;
}
$hello('World'); //output: Hello World
```

This is not sheer madness; of course it has purpose. Many functions in PHP take functions as arguments, so it comes to help in those situations.

Consider a situation like this:

```
function yourName($value){
//do something
}
array_map ('yourName', $names);
```

But instead of that, you just use an anonymous function in-line and define it like this:

```
array_map(function($value){
//do something with $value
}, $names)
```

Finally, an anonymous function can be used as closures in PHP, a fairly advanced concept but less common in PHP; it is pretty often used in JavaScript. And in Laravel you have already found it in the route-level usability of an anonymous function.

```
Route::get('/', function(){
return 'Hello World!'
});
```

CHAPTER 5



Controller Class

In MVC framework controller has a definite role. It primarily controls the flow between model and view. In 'routes.php' file we've seen how we send the request using HTTP protocol. Now we want to organize the same behavior using controller class.

Let us remember the first route method. When you install Laravel it normally comes with this default route method.

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

In this route method we get a view. As the route method says: return a view. The question is: where are models and controllers? We are not supposed to create only views using a Closure, are we?

Suppose we have 100 pages in our application. We can't make a route for every single page. We can't put all our HTML and CSS codes in that 'routes.php' file. That design is simply ugly and unthinkable. We shouldn't put our application logic in the 'routes.php' file either. To solve this problem the concept of controller class enters the scene.

Controller means controlling the application. It's quite literal: it controls or manages the application layers. What are the layers?

A nicely designed application always has some hidden, inner layers. Users should not get any hint of it. All they should see is the presentation layer. Application logic should always stay hidden.

As a beginner, you should get accustomed to this layer division. A good object-oriented programming sense will always guide you to separate the application logic from the presentation logic.

A controller is a transporter. It transports application logic to the presentation layer. A good design principle always encourages you write your application logic in a separate hidden file. A controller will only execute those methods and return a view. It will never know what it carries.

To start with let us quickly create a controller. It's a good practice that you should plan it first. Suppose you want to make a simple controller that will only return a view. How about giving a name 'MyController'? We keep this controller in 'app/Http/Controllers' folder.

Here is the code of our MyController.php file:

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
use App\Http\Requests;

class MyController extends Controller
{
    // returning a simple page using view
    public function returningASimplePage() {
        return view('newpage');
    }
}
```

I have written this code manually on my text editor. Actually you can create any controller automatically through command line using 'artisan' command. We'll learn about 'artisan' command in detail later. After that we will also learn about models and views and connect them together to get a total understanding of model, view, controller flow. In fact, later you'll find more usages of the 'artisan' command in creating database migrations, controllers, models, and many more. Just go to your Laravel folder and issue this command: `php artisan`. Then, observe the output.

Now it's time to use this controller in our 'routes.php' file. Till now, we have only used routing to go to the view page. A view page is actually a presentation layer. Let us think in an object-oriented way.

For the first time we're going to use something different: a controller. In the 'routes.php' file we add this code:

```
Route::get('newpage', 'MyController@returningASimplePage');
```

Let us understand this line word by word. The first part is quite okay. Route class uses a static method giving a URL: 'newpage'. In the second part we write the name of the controller 'MyController' and use a special '@' sign to call the function 'returningASimplePage'.

If you go back and have a look at your MyController code you'll find that this 'returningASimplePage' function simply returns a 'newpage'. We have already stored this 'newpage.blade.php' file in our 'views' folder before. I don't want to give that long HTML code. It's almost same as the HTML code we had used in our 'routes.php' file before.

Now we may type <http://localhost:8000/newpage> to see what it shows! (Figure 5-1.)

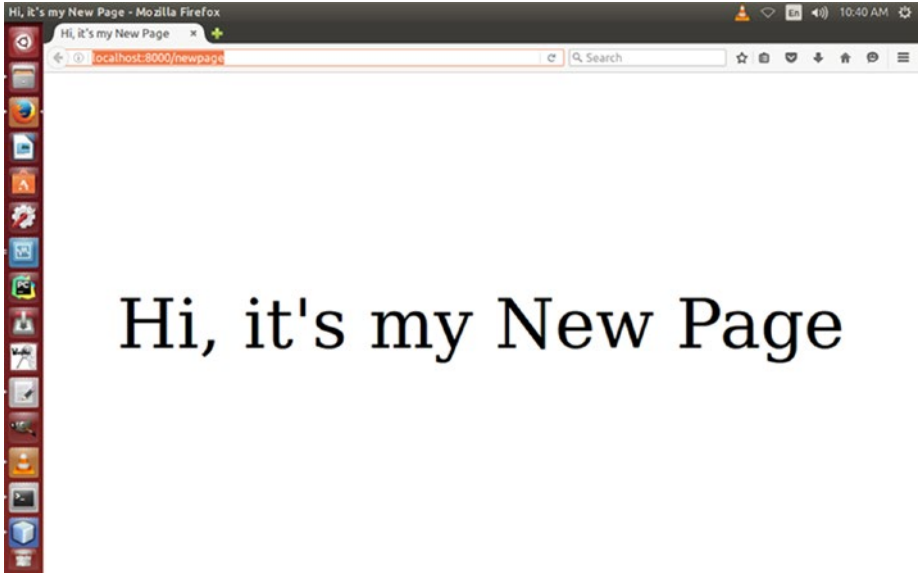


Figure 5-1. A simple page view through a simple controller

It's a nice little output and we have produced this output from a controller. We've progressed a little bit!

As you have just seen in 'MyController.php' file, it's a class. It must have a public function; otherwise, Laravel can't route it.

While writing a controller class you need to remember few things.

First, we render a view using a controller class always using public functions.

```
public function index() {
    return view('home');
}
```

Second, in the 'routes.php' file, we no longer use the anonymous function or Closure. We route the controller instead.

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

In this first part of controller class, we're not diving in too deep. Let us first see how model and view work in unison with a controller class. After that, in the second part of controller class, we'll see more functionality and try to build up a dynamic database-oriented application.

5.1 RESTful Controller

Before we start this chapter, let's realize one thing. REST (Representational State Transfer) is an architectural protocol through which we imagine our URLs as nouns and give them common verbs like GET, POST, PUT, or DELETE. Every URL is a representation of an imaginary resource. I hope I can discuss this interesting topic in the sequel to this basic Laravel Learner book. As I used this term before, I assume you have already become familiar with the term 'RESTful'. What does it mean? In a simple sense it offers some solutions. When we route to Closures we can define the HTTP verb in the method form.

```
Route::get('index', 'Customer\Profiling\ProfileController@showIndex');
```

But we are now concerned with Controller, so we must get a solution for that also. Hence, 'RESTful' comes to our rescue. Now we can explicitly tell our transport mechanism: what kind of verbs we should use and how. How we can do that? 'app/Controllers/profile.php' can be written like this:

```
namespace Customer\Profiling;

class ProfileController extends BaseController
{
    public function getProfile()
    {
        return View::make('profile');
    }
    public function postCreate()
    {
        //some form here to collect data
    }
}
```

These methods could have been more. The number really does not matter. We can add 'delete' verb also. Now in our 'routes.php' we can use all those HTTP verbs in a single command like this:

```
Route::controller('profile', 'Customer\Profiling\ProfileController');
```

Through this command we can route all HTTP verbs used in the 'ProfileController' class. Our intention is pretty simple. We tried to show up a form where users will fill up their names and other details. After that we take up those data and post them to our database. In Laravel 3.2, we used to write it differently, but the 'get' or 'put' prefixes are there also.

5.2 Role of a Controller

The very first question is the following: what is a controller? As you have seen in the examples given so far, it is nothing but a kind of transport mechanism. It takes your users or viewers to the presentation layer. We remember the law that a class should have a singular task. The Controller class takes you to the presentation layer and its job should be finished there. It should not know your application or domain logic. Why not? Because it is a transport mechanism, nothing else. It is like a data cable to a TV monitor. There are plenty of other things in your application going on in the background, and you should take care of that so that an outsider could not reach there. Imagine a situation like this:

```
class UserController extends BaseController {
  public function getIndex()
  {
    $users = User::all();
    return View::make('users.index', compact('users'));
  }
}
```

This ‘User::all()’ refers to Eloquent ORM proposition. Don’t worry, we will come to that chapter very soon in the database part and will discuss about ‘Eloquent ORM’ in great detail. Presently you need know only that it keeps contact with the data layer. Now, should we connect to the database directly from our Controllers? Should a transport mechanism which is supposed to communicate with our presentation layer also talk to the data layer? A data layer is a vulnerable spot, and it will put us in a tight spot if someone puts some dirty codes into it. Remember, controller is a carrier, so it will carry it back to the data layer and we will have to face the mess!

So, it’s better to keep it in the dark about the domain logic or data layer and make its job singular by decoupling it in a way suitable to our application. How we can do that? Here, Interface comes to our rescue: a great thing indeed. In the Interfaces chapter we talked about coming back to it, and so here we are once again. Interfaces can save us by encapsulating the actual logic. We are safe. As we don’t want to hit the database directly from our Controller class, we keep a shield in between them, namely, Interfaces. It will act as a medium between Eloquent ORM and Controller. Our Controller will never know from where the data comes. It will do its job and forget about it. It has many advantages like Abstract classes. But there is a subtle difference between them. An Interface defines the syntactical contract that all the derived classes should follow. Specifically, the interface defines the ‘what’ part of the syntactical contract and the derived classes will describe the ‘how’ part of the contract.

```
interface OrderDetails {
  public function UpdateCustomerStatus(array
$user, $amount);
}
class UserClass implements OrderDetails {
  public function UpdateCustomerStatus(array
$user, $amount) {
    //do something through Eloquent
  }
}
```

You can use several interfaces between them; don't worry about using them throughout your application. The more you use it, the better. Some people may object to the extra typing, but personally I think that using Interfaces would make your application more robust and agile. You can test them without hitting your domain logic from your web layer.

5.3 Resourceful Controller

Resourceful controller is one of the greatest new controller staffs that Laravel 4 comes with. The resource method of controller class sometimes can do the magic for us, and we are going to see how it takes place! Let's take a look at my artisan command and find if there is anything about the Controller staff. Type the following:

```
php artisan -list
```

And we see a controller:make method to make a resourceful controller. Very nice to have found out that in a much improved php artisan command line we can make resourceful controller.

Let's create a resourceful controller using the php artisan command and see what happens. We can issue the command like this:

```
php artisan controller:make UserController
```

Now if we leave it that way, it will create a resourceful controller in my project's controller folder. The sky's the limit for what you can do with your UserController resourceful controller. You can show any user with its ID, you can edit, update and create, and even destroy or delete the records. Presently we are interested in the first two methods, that is, index and show(\$id). Let us declare a new Route::resource() method in our routes.php file. It is something like this:

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

As you see, we can call our resource 'sanjib' and that resource is handled by the 'UserController' and that only handles the 'index' and 'show' method.

Now if you type <http://localhost:8000/sanjib> you see a nice output of anything that you return from your 'UserController' index method. And besides, if you type <http://localhost:8000/sanjib/1>, it will give a nice output with the passing \$id, which is 1 here.

So this is a great step forward from Laravel 3, where this '\$id' passed as a method. Anyway, if you write like Route::resource('sanjib', 'UserController'), it will produce all seven methods I have just shown you. And finally you can create, store,

edit, update, and delete your users from these methods. Now we can attach filter to every single method either at one go or singlehandedly. Let us make a change in our UserController code, adding a line of calling constructor filter like this:

```
//UserController.php
public function __construct() {
    $this->beforeFilter('sanjib');
}
```

And with it we are going to attach filter to all of my methods in 'routes.php' file like this:

```
Route::filter('sanjib', function(){
    return 'This filter is attached to every
method of resource sanjib';
});
```

Now we can restrict this filter to the only 'index' or any method you wish. That is also a piece of cake now. If you want to attach it to the 'index', just write down the following:

```
public function __construct() {
    $this->beforeFilter('sanjib', ['only' => ['index']]);
}
```

That way, only 'index' method is filtered, nothing else. I hope the role of a controller is pretty clear now and we can do some tweaking with the dynamically built-up 'View' page and see how we can control and pass our data through controller and build a simple web site.

5.4 Controller, IoC Container, and Interface

Till now, we have talked a lot about what a Controller should know. But, you may ask, where is its implementation? Exactly! That is what I am going to show you now and this is the pure beauty of Laravel 4.2's code-happy approaches. And we all want to be code happy, don't we? We should be able to write some decent code: if we're not as accomplished as Taylor Otwell, at least we could try some fancy stuff that would satisfy our in-born aesthetic.

So let us try some good practices. As we wanted to limit the knowledge of a controller, we implement an interface and make our controller ignorant of any other activities except the transport mechanism. If inside our UserController we write something like this:

```
$user = User::all();
return View::make('user.index')
>with('user', $user);
```

our controller directly hits the database. That we don't want. So what can we do instead?

Let us create an interface 'AllUser' in our model folder. The code is simple.

```
<?php
/*
 * this interface gets all users through a class
 called GetAllUsers
 */
interface AllUsers {
    public function getUsers();
}
```

This interface will work as a contract between our classes 'GetAllUsers' and our controller 'UserController'. Let us create a class 'GetAllUsers' like this:

```
<?php
/*
 * this class implements AllUsers interface and
 bound in IoC Container
 */
class GetAllUsers implements AllUsers {
    public function getUsers(){
        $users = User::all();
        return $users;
    }
}
```

Through this class we retrieve all of our users from the database table users. We still did not cover view, model, Eloquent ORM, and database, so don't worry. All you need to know now that we have a user table inside our database and we are retrieving all staff from the users table.

Now finally how does our controller 'UserController' look like?

```
<?php
class UserController extends BaseController {
protected $users;
    public function __construct(AllUsers
$users) {
        $this->users = $users;
    }
    public function getUsers(){
        $user = $this->users->getUsers();
        return View::make('user.index')->with('user', $user);
    }
    public function index()
{
//
```

```

}
public function create()
{
//
}
public function store()
{
//
}
public function show($id)
{

//

}
public function edit($id)
{
//
}
public function update($id)
{
//
}
public function destroy($id)
{
//
}
}

```

This part is particularly important:

```

protected $users;
public function __construct(AllUsers $users) {

    $this->users = $users;
}
public function getUsers(){
    $user = $this->users->getUsers();
    return View::make('user.index')->with('user',
    $user);
}
}

```

We have implemented the interface and through our class 'GetAllUsers' we finally got the whole users table. Now in 'index' View we just loop through the 'user' object and extract the 'username' like this:

```

foreach ($user as $value) {
    echo $value->username . "<br>";
}

```

Now the logic is clear and very lucid to follow. Now we are free to change the data target any time just changing inside the class or creating another class. Here we have used 'MySQL' but we can change it any time to any database you like. The most important thing is that our controller would never know what is going on inside. But the final hacking lies inside our 'route' file, because we have to bind the class and the interface and the great IoC container plays the pivotal role here. This is very simple with your 'App' Facades. The code is as follows:

```
App::bind('AllUsers', 'GetAllUsers');
```

As you see, our 'App' Facades bind the class and interface together so that a nice layer has been included between our data source and the transport mechanism. Moreover, our controller does not know about what has been going on inside that layer. Our application becomes more decoupled and you easily test it with 'phpUnit' without directly hitting the database.

5.5 Summary

We normally organize our transport mechanism or web layer through controller class. So this is a good practice to make it ignorant about our domain logic and use interfaces as a medium between web and data layer. Through controller classes we organize our route-level logic and besides we can use modern features like dependency injections.

Use interface and let controllers maintain their singularity of job which should be concerned only with the web layer. Since we use Composer to autoload our PHP classes, it can live anywhere in the file system as long as controller knows about it. And routing to controller is entirely decoupled from the file system.

Resourceful controllers usually make RESTful controller around resources.

Finally, we can conclude with the filter part, which is extremely important in managing the administration of an application. You can either control it through route level or explicitly use it inside your controller.

```
Route::get('profile', array('before' => 'auth',
'uses' => 'UserController@showProfile'));
```

Or you may specifically mention it inside the controller:

```
Class UserController extends BaseController {
public function __construct()
{
$this->beforeFilter('auth', array('except' =>
'getLogin'));
$this->beforeFilter('csrf', array('on' =>
'post'));
$this->afterFilter('log', array('only' =>
array('fooAction', 'barAction')));
}
}
```

And finally filter can be used through constructor like this:

```
class UserController extends BaseController {
public function __construct()
{
$this->beforeFilter(function()
{
// some code
});
}
}
```

After mastering the basic controller, it is good to consult the Laravel official documentation to hone your skills.

CHAPTER 6



View and Blade

In the MVC framework, the part 'view' has an important role. First of all it should look good and appealing. It deals with HTML, CSS, JavaScript, and many other codes that make a page look great.

At present a little knowledge is necessary. You need not be an expert to learn this chapter. 'Views' folder stays inside 'resources' folder. You need to keep your html codes here. To make it look awesome, you may wish to have a great CSS style! That CSS file and other necessary JavaScript files should go to inside 'public' folder. Laravel connects them easily.

We'll learn this process step by step.

First we'll create few static pages. Second we'll pass dynamic data so that we can later build a dynamic application on the top of that.

Suppose we want to have an 'About' page in your application. Let us create it and connect it to our 'MyController' and finally route it to the browser.

It's quite natural that we won't stop at having only one static page. Later we might need a 'Contact Us' page or any number of other pages. Keeping that perspective in mind let's create a folder 'pages' inside 'views' folder. In the 'pages' folder we first create a 'master.blade.php' file. This is a master layout page. It's not compulsory that we should always call it a 'master'. We can call it 'layout.blade.php' or 'default.blade.php' or we can choose any other meaningful name. This master page is a blueprint of our basic layout that other pages will follow.

Remember, you can always create separate folders to keep separate master layouts for other pages. It's needless to say that you should always have only one master layout for your entire application.

Let us keep our master layout simple enough. Before going to write the codes, we may want to learn one thing: what does the term 'blade' mean?

Well, 'blade' is the template engine of Laravel and it should be written in php. It has its own functions to make your life easier. And for that reason you need to add an extension of 'blade.php'.

Let us write a simple 'master.blade.php' code and it'll have some special 'blade' templating functions.

```
//resources/views/pages/master.blade.php
<!DOCTYPE html>
<html>
  <head>
```

```

        @yield('head')
        <link rel='stylesheet' href='/css/style.css'>
    <title>
        @yield('title')
    </title>
</head>
<body>
<div class="container">
    <div class="heading">
        @yield('heading')
    </div>
<div class="content">
@yield('content')
    </div>
<div class="footer">@yield('footer')</div>
</div>
</body>
</html>

```

As you see, we have used only one function— 'yield'—inside the master page. We call that function with a '@' sign. This function has a literal meaning. It really yields or produces or generates something.

We have segmented our HTML codes into a few parts so that they can be easily remembered. In the 'head' part we have two 'yield' functions: 'head' and 'title'. Under the '@yield('head')' function we have our CSS style link and the other 'yield' function has 'title' inside it. The rest part follows the CSS style division classes. In the 'heading' we keep a large point heading. In the content part we have defined body point, font style, et cetera.

Let us see our CSS style code that we have kept in the 'public/css' folder.

```

//public/css/style.css
html, body {
    height: 100%;
}
body {
    margin: 10;
    padding: 10;
    width: 90%;
    display:compact;
    font-weight: 100;
    font-family: 'Lato';
    font-size: 22px;
}
.container {
    text-align:left;
    display: table-cell;
    vertical-align: middle;
}

```

```

        .content {
            text-align:left;
            display: inline-block;
        }
        .footer {
            text-align:right;
            display:compact;
            font-family: 'Lato';
            font-size: 14px;
            color: red;
        }
        .heading {
text-align:center;
            font-size: 136px;
            text-height:max-size;
            color:indianred;
        }
        .h1 {
            text-align:center;
            font-size: 66px;
            font-family:fantasy;
            color:tomato;
        }
        .h2 {
            text-align:center;
            font-size: 46px;
            font-family:cursive;
            color:coral;
        }
    }

```

So far, we have defined our ‘master.blade.php’ file and ‘style.css’ file. You have probably noticed that in the ‘master.blade.php’ file in the ‘head’ section we give the CSS style link. Now there is an option where you can place this CSS file link in your ‘about.blade.php’ page.

Let us first write down the ‘about.blade.php’ file.

```

//resources/views/pages/about.blade.php
@extends('pages.master')
@section('head')
@stop
@section('title')
    It's a test page
@stop
@section('heading')
    {{ $name }} {{ $profession }}
@stop
@section('content')
    <div class='h1'>

```

```

    Every Writing is a Problem!
</div>
<p>
    There was a problem few minutes back. There was a problem few minutes
    back.
    There was a problem few minutes back. There was a problem few minutes
    back.
    There was a problem few minutes back.
</p>
<div class='h2'>
    I have just solved it.
</div>
<p>
    Have a nice time folks.
</p>
@stop
@section('footer')
    Home of Sanjib Sinha
@stop

```

At the top we use the function ‘extends’, which literally extends the modularity of the master page. After that, we continually use only two functions: ‘section’ and ‘stop’. These functions actually connect to the CSS styles you have defined earlier.

Each section has its own division: ‘head’, ‘title’, ‘heading’, ‘content’, and ‘footer’. Each ‘section’ contains its own contents and then we use ‘stop’ function to stop that ‘section’.

Inside the ‘head’ section of ‘master.blade.php’, we have used the CSS style link. You can also use the same link inside the ‘about.blade.php’. In that case, you need not use the CSS style link inside the ‘master.blade.php’ file. The top part of our ‘about.blade.php’ page turns out to be like this:

```

//resources/views/pages/about.blade.php
@extends('pages.master')
@section('head')
<link rel='stylesheet' href='/css/style.css'>
@stop

```

But using CSS style inside ‘master.blade.php’ is always preferable.

Now, hopefully you can make out a relationship between the functions of ‘master.blade.php’ and ‘about.blade.php’. The master page plays the role of a parent and the about page is its child. The logic flows down from parent to the child.

In the ‘about.blade.php’ we encounter two foreign terms in the ‘heading’ section:

```

//resources/views/pages/about.blade.php
@section('heading')
{{ $name }} {{ $profession }}
@stop

```

You are smart enough to guess that they are the names of the variables that we have used in our controller. Otherwise, how they'd appear in our 'about.blade.php' files? We have not seen them in our master page.

You have guessed right: they come from the controller, 'MyController.php' file. Earlier, we have talked about passing data dynamically through controller to the view blade. Since it's an 'about' page we can pass a few pieces of interesting data about us.

We have decided to pass two important data points through the controller: 'name' and 'profession'. Let us see the MyController code first.

```
//app/Http/Controllers/MyController.php
public function about() {
    $name = 'John Doe, ';
    $profession = 'A Writer';
    return view('pages.about', compact('name', 'profession'));
}
```

You see that we have defined two variables first and then return those variables using 'view' method to the 'pages.about'. Here the term 'pages' refers to the folder name 'resources/views/pages'. The second part—'about'—refers to the 'about.blade.php' file. We could have written it as 'pages/about' in MyController. But I personally prefer the '.' notation as it reflects the object-oriented approach.

This passing of data can be done using many tricks. We have shown the PHP 'compact' method. But it could have been done this way also.

```
//app/Http/Controllers/MyController.php
public function about() {
    $name = ' John Doe ';
    return view('pages.about')->with('name', $name);
}
```

Or we could have passed it as an array value, and you can pass a long array this way.

```
//app/Http/Controllers/MyController.php
public function about() {
    return view('pages.about')->with([
        'name'=>'John Doe',
        'profession'=>'A Writer'
    ]);
}
```

Finally, we can check how it looks in our favorite browser. Type <http://localhost:8000/about> and it opens up the page (Figure 6-1).

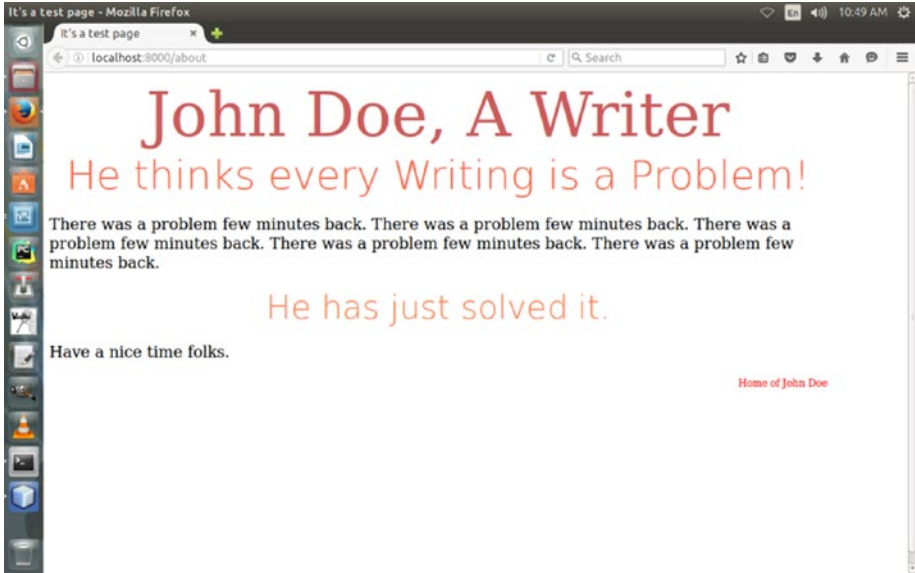


Figure 6-1. We have dynamically passed data to view

So far, we have learned a few tricks that will enable us to create static pages with some dynamically generated data through the controller.

Having progressed a little bit, we have gained some confidence, and we can probably venture out now to take on some more difficult tasks.

CHAPTER 7



Environment

Laravel comes with many stunning features. One of them is definitely database migrations. In the next chapter we'll discuss migration in detail.

Before that we need to understand our environment properly. What is environment in Laravel? If you look at the document root you will find an `.env` file. It basically tells us about the database connections. Where would we get that default database setup file? In the `config` folder, we have a `database.php` file. Open it up and see the content.

We see a line that tells us about the default database connection.

```
//config/database.php
'default' => env('DB_CONNECTION', 'mysql'),
```

It's MySQL. We're not going to change it for now although it's temporary, because in the later part of the book we'll see how we can work with SQLite database. At present just keep it as it is - our default database is MySQL. Now opening up our `.env` file we add our database, username, and password.

```
 //.env
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=testdb
DB_USERNAME=root
DB_PASSWORD=pass
```

There is a catch, of course. Suppose we'd like to place our project in any cloud repository like `github`. In that case, our secrets may come out.

Laravel has thought this through.

In the `.gitignore` file, it has already added this line:

```
 //.gitignore
/vendor
/node_modules
/public/storage
Homestead.yaml
Homestead.json
.env
```

When you place the whole project, it'll automatically ignore the listed folders and files. `Homestead.yaml` and `Homestead.json` are among them.

We've set up our environment and now we can safely move to our next big chapter: database migrations. After that we'll discuss Eloquent, and then we'll finally see how Model, View, and Controller workflow works. We'll also see how we can "create, retrieve, update, and delete" our SQLite database.

CHAPTER 8



Database Migration

Database migration is one of the best features that Laravel provides. As a beginner, you may find acquiring the concept a little bit difficult. But once you have learned about database migration, the experience of working with any kind of database becomes extremely easy and pleasant.

In the past you need to create a database, table, and columns either by SQL coding or by using a tool like PHPMyAdmin. The task seems to be daunting sometimes. With Laravel, it's no longer difficult.

Besides, we have an extra advantage. As a developer, working in a team, you may get your database job well synchronized with your colleagues. In that sense it's become like a version control. You can easily roll out your database table and update with new features through php codes inside your application.

All you need to do is write a few lines of PHP code and Laravel will look after the next steps. It's that simple.

Let's start with a simple 'tasks' table.

We've already mentioned the database names and other stuff in our '.env' file. Now it's time to create a 'task' table and add some columns to it.

Before starting a new migration let us go to our 'database/migrations/' folder; we find that two PHP files have already been stored there. These migration files are 'users' and 'password resets'. Laravel comes with them so we can have a look and try to understand how it works actually.

Open the 'users' table. It has a long name: '2014_10_12_000000_create_users_table.php'.

```
//database/migrations/2014_10_12_000000_create_users_table.php
<?php
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateUsersTable extends Migration
{
    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();
    });
}
public function down()
{
    Schema::drop('users');
}
}

```

As you see, there are only two functions: 'up' and 'down'. The instructions written inside are fairly simple to understand. It tells us about creating columns in 'users' table. It mentions the characteristics of the columns: whether it'll be 'string', or 'text', or have any extra feature of uniqueness. The method 'up' means you generate or update the tables. The method 'down' has a significant meaning. It means you can roll back your migrations any time, and generate it again.

Suppose in the 'tasks' table, you suddenly think about changing the name of a certain column. You can roll back your old migration. You can update it in your migration PHP file and then you can run the command again.

Let's start creating our new 'tasks' table. Open up your terminal and issue this command:

```

//making migration
php artisan make:migration create_tasks_table --create="tasks"

```

Once you have issued this command, it'll automatically run the migration and it'll show it up on your terminal.

After creating the 'tasks' table, it'll also show you the other migrations that have been shipped with Laravel.

```

//run the migrate command
php artisan migrate

```

After a successful migration it will show all migrations at one place.

```

//showing the migrations
Migration table created successfully.
Migrated: 2014_10_12_000000_create_users_table
Migrated: 2014_10_12_100000_create_password_resets_table
Migrated: 2016_08_30_024812_create_tasks_table

```

Now we actually have three tables at hand. The new table 'tasks' has also been created successfully. You can either check it on your terminal or open up your MySQL tool and see it. Besides, we need to see how the migration PHP file has been created in our 'database/migrations' folder.

There should be a file called '2016_08_30_024812_create_tasks_table.php' in that folder. Let us open it see what it has in store for us.

```
//database/migrations/2016_08_30_024812_create_tasks_table.php
<?php
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateTasksTable extends Migration
{
    public function up()
    {
        Schema::create('tasks', function (Blueprint $table) {
            $table->increments('id');
            $table->timestamps();
        });
    }
    public function down()
    {
        Schema::drop('tasks');
    }
}
```

You see the difference between the default 'users' table and the newly created 'tasks' table. Our 'tasks' table comes up with two methods—as usual, 'up' and 'down'—but it has only two columns: 'id' and 'timestamps'.

Remember, whenever you run a migration, a PHP file is generated inside 'database/migrations' folder and it looks like this. Now, it's our responsibility to add columns into it and again issue the migration command. We did the same thing here. We have changed this part. We've added title and body columns to complete our 'tasks' table.

```
//database/migrations/2016_08_30_024812_create_tasks_table.php
public function up()
{
    Schema::create('tasks', function (Blueprint $table) {
        $table->increments('id');
        $table->string('title');
        $table->text('body');
        $table->timestamps();
    });
}
```

We run 'migrate' command again to update the database table. After that it automatically updates itself.

```
//run the migrate command
php artisan migrate
```

In this step, suppose we have forgotten a column in the newly created 'tasks' table. Can we add it after all the migrations are over? Yes, we can do that.

The greatness of Laravel is it has already thought about every possibility that might happen. This time the command is slightly different.

```
//adding a new column to the table  
php artisan make:migration add_reminder_to_tasks_table --table="tasks"  
Created Migration: 2016_08_30_035529_add_reminder_to_tasks_table
```

We forgot to add a column 'reminder' into our 'tasks' table. We have just added it using migration command. Now we can have a look at our newly created 'tasks' database table in our phpMyAdmin tool (Figure 8-1).

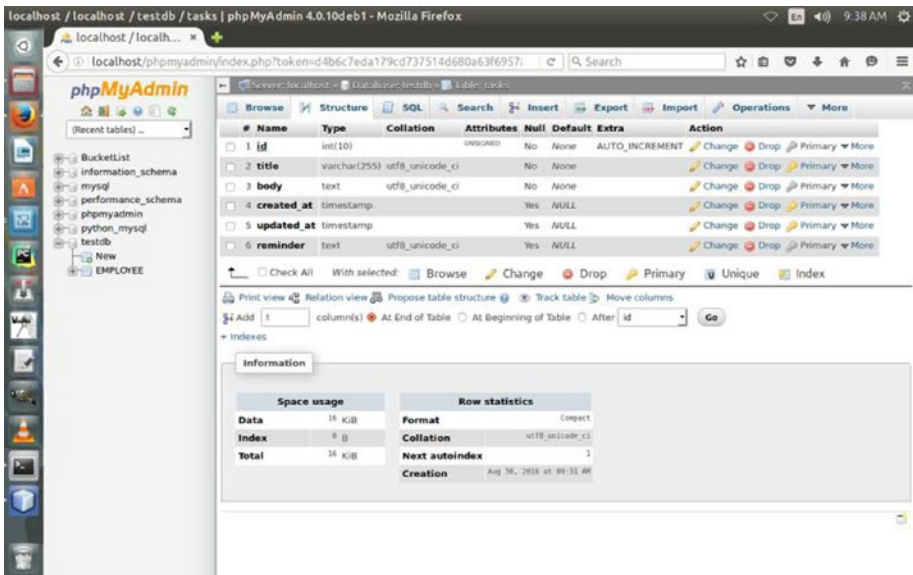


Figure 8-1. The 'tasks' table at phpMyAdmin tool

Our 'tasks' table has all the columns now: 'id', 'title', 'body', 'timestamps', and finally 'reminder', which we added later.

8.1 Summary

Database migration in Laravel is a technique that helps us to create, update, or delete the table and its columns. In a developmental scenario, when you work on a project participating from different destinations, you can easily modify or manipulate your database through a few simple php commands.

CHAPTER 9



Eloquent

Eloquent Model class is used for Laravel's 'active record implementation' methods. It literally means that we can easily 'CREATE, RETRIEVE, UPDATE, or DELETE' any database record using this Eloquent Model class.

In the Laravel PHP artisan command we generate files through 'make'. We have seen it before in case of 'controllers', 'migrations'. Now we can do the same thing in our Model class.

Remember one thing: if you have a table called 'tasks', then you must have a model called 'Task'. If we have a table called 'songs', we must have a model called 'Song'. In the model, the name starts with the capital letter and the last letter of the table is ignored.

Let us create our first model class 'Task'

As usual we open our terminal and issue this command:

```
//creating model task
php artisan make:model Task
Model created successfully.
```

Once we have issued this command, in the 'app' folder a PHP file is automatically generated.

```
//app/Task.php
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
class Task extends Model
{
}
//code ended
```

It's a simple PHP file with an empty class and you can't realize its power at one glance.

But the reality is different. The 'Task' class extends 'Model' class, which is a fairly complex class with every kind of database functionality. It has many complex methods that actively implement many things. To name a few, there are 'save', 'find', 'update' methods and many more. You can save your record. You can update your record. You can issue a mass assignment. There is more functionality that we are going to see in the future course.

Let us first fill our database table 'tasks' with a few tasks.

To do that we can either add or update records through an administrative dashboard that has forms to take inputs or we can use any other trick.

At present we've not learned anything about the usages of forms. We'll learn it later. So we can use 'Tinker' on our terminal to add a few quick records to test our 'Task' model.

What is 'Tinker'?

It's an artisan command that comes with Laravel and allows us to work directly on Laravel code base through terminal.

If we issue this command: `php artisan tinker`, it just opens up a nice interface on the terminal to work ceaselessly.

```
//adding to database table 'tasks' by tinker
php artisan tinker
Psy Shell v0.7.2 (PHP 5.5.9-1ubuntu4.14 - cli) by Justin Hileman
>>> $task = new App\Task;
=> App\Task {#632}
>>> $task
=> App\Task {#632}
>>> $task->title = 'My First task';
=> "My First task"
>>> $task->body = 'I will wake up early in the morning tomorrow.';
=> "I will wake up early in the morning tomorrow."
>>> $task->reminder = 'Do not forget';
=> "Do not forget"
>>> $task->toArray();
=> [
    "title" => "My First task",
    "body" => "I will wake up early in the morning tomorrow.",
    "reminder" => "Do not forget",
]
>>> $task->save();
=> true
>>>
//adding completed
```

Actually we have created a 'task' object using the 'Task' class. Now that object can automatically access the columns almost like properties. Watch this part of the previous tinker operation.

```
>>> $task->title = 'My First task';
=> "My First task"
```

It actually keeps those values in memory. When you finally call the `save()` method, it saves all the records in the table.

Let us fill with another set of records.

```
//create new set of record
php artisan tinker
```

```

Psy Shell v0.7.2 (PHP 5.5.9-1ubuntu4.14 - cli) by Justin Hileman
>>> $task = new App\Task;
=> App\Task {#632}
  >>> $task->create(['title' => 'My Second Task', 'body'
=> 'I would like to get up early in the morning',
'reminder' => 'I need to buy medicines']);
  => App\Task {#643
    title: "My Second Task",
    body: "I would like to get up early in the morning",
    reminder: "I need to buy medicines",
    updated_at: "2016-08-30 04:45:10",
    created_at: "2016-08-30 04:45:10",
    id: 2,
  }
  >>>
//created successfully

```

Can we update the first record now? Yes, of course we can.

There are two methods through which you can update your previous records. The first method is the old method of writing it again and calling the save() method like before.

```

//now you want to update the first record
>>> $task = App\Task::find(1);
=> App\Task {#631
  id: "1",
  title: "My First task",
  body: "I would like to go to market by nine.",
  created_at: "2016-08-30 04:20:53",
  updated_at: "2016-08-30 04:37:52",
  reminder: "You must go. You need to buy medicines.",
}
>>> $task->body = 'I will wake up in the morning.';
=> "I will wake up in the morning."
>>> $task->save();
=> true
  >>> $task->reminder = 'I must do.';
  => "I must do."
  >>> $task->save();
  => true
  >>>
//record successfully updated

```

There is another method where you can call update() method directly to update your record.

```

//another update method
>>> $task->update(['reminder' => 'I must do and not forget!']);

```

```
=> true
>>>
//record successfully updated
```

Finally we want to test a mass assignment using eloquent model class.

```
>>> $task = new App\Task;
=> App\Task {#633}
>>> $task->create(['title' => 'My Third Task', 'body'
=> 'I have to start car engine', 'reminder' => 'Battery
will choke']);
=> App\Task {#646
    title: "My Third Task",
    body: "I have to start car engine",
    reminder: "Battery will choke",
    updated_at: "2016-08-31 05:36:16",
    created_at: "2016-08-31 05:36:16",
    id: 3,
}
>>>
```

It didn't give any error. But it would have given the error had we not already changed the eloquent 'Task' model class.

We had earlier added this line into that empty class.

```
//app/Task.php
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
class Task extends Model
{
    protected $fillable = [
        'title',
        'body',
        'reminder'
    ];
}
```

We have added a property `$fillable` and we instruct Laravel Task model class to remember that three columns—'title', 'body', and 'reminder'—should be fillable through mass assignment. These columns may be mass assigned at any time.

Finally, we can conclude that after migrating the database, eloquent model class actually allows us to do database operations more flexibly. But it is not the end; rather it's a great start. Eloquent model class has many more surprising gifts for developers. Now we'll learn them through a complete Model-View-Controller workflow.

To do that, we will create a new controller and a few view pages that will render the database output. We've already created a model—'Task'—so we need not create that again. And using artisan command 'tinker', we have added two or three records into our MySQL database table.

We'll use the controller class to use the eloquent model class abstraction in turn to produce a nice-looking view of our database table on the browser.

It will look like the image in Figure 9-1:

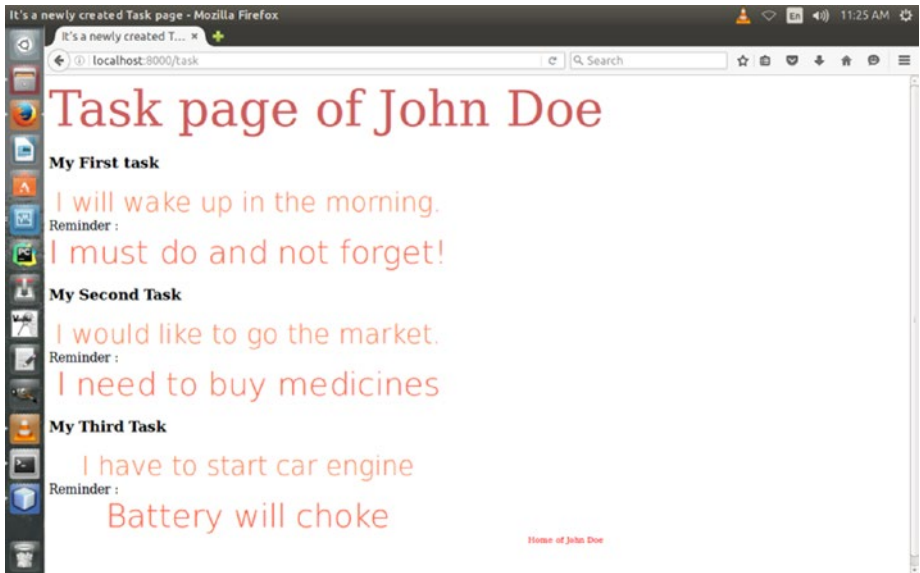


Figure 9-1. A view of database table 'tasks'

This view is a product of a Model-View-Controller workflow. In the next chapter we'll see how it works.



Model, View, Controller Workflow

So far we have learned a few important things, including routing, controller, view, database migrations, and eloquent model. Actually we've learned a lot and done a lot of things that cannot be done in such a short period of time with such small lines of code.

We've used MySQL database to build a simple task table where we can now create, retrieve, update, and finally delete our tasks using `terminal` on our terminal.

In fact we've done a lot of things quite easily with less code. You've built a full-fledged CRUD (create, retrieve, update, delete) application without writing a single line of SQL code.

If you didn't use Laravel you would have to write several hundred lines of php and sql codes. Laravel has done it in an object-oriented way. If you wanted to do that single-handedly, you need to create at least 25 php files to get that taste of layer divisions. Moreover, you would have to use a MySQL tool like phpMyAdmin. Now it may seem cumbersome.

What is the most striking feature that we've discovered so far?

What strikes me most is the universality of this application. Many things have been housed under one roof. You don't have to go outside your application to do anything. You can do the database migrations by creating, updating, or deleting database tables with the help of just a few lines of php code. You can create, update, or delete data into the database with a few lines of php code without worrying about any MySQL tool. Everything is so simple.

Staying inside your application and doing difficult things without going outside are no joke. Laravel has done it for us.

Now we have a full database table that has stored few tasks. Next, we'll see how easy it is to retrieve that data and present on a web page. All we need is a controller, a view page, and a model.

Actually, we already have a model called 'Task', so we need not create it anymore. Let us build the controller class first that will fetch the data from database on a view page.

Let us register the controller first in our 'routes.php'.

```
Route::get('task', 'TaskController@index');
```

Now we can create a controller using artisan command. Let us do it.

```
php artisan make:TaskController
```

It creates a controller class—a php file—‘TaskController.php’, in ‘app/Http/Controllers’ folder. It’s a simple class without any methods defined. Let us create a method called ‘index’ that will fetch all the records from our ‘tasks’ table in the database.

To do that we need our eloquent model class ‘Task’ that we’ve already defined in the previous chapter. We need to use that class on the top of the ‘TaskController.php’ file and just call the ‘all()’ method from it.

```
$tasks = Task::all();
return $tasks;
```

It looks pretty clean and simple. The output comes in ‘json’ like this:

```
[{"id":1,"title":"My First task","body":"I will wake up in the
morning.","created_at":"2016-08-30 04:20:53","updated_at":"2016-08-30
04:53:22","reminder":"I must do and not forget!"}
.....
//the rest is omitted for brevity
```

Laravel knows that the output from a database might be used for any API. So the ‘json’ format is the answer to it.

In just two lines of code we retrieved all our data from a database table. But, we want to get that data on our view page. We need to create a view page. We have learned how to do that. We’ve also learned how to pass data from a controller to a view page.

So finally, our TaskController class stands out with clean and simple codes that fetch all data from the database using eloquent ‘Task’ model class.

```
<?php
namespace App\Http\Controllers;
use App\Task;
use Illuminate\Http\Request;
use App\Http\Requests;

class TaskController extends Controller
{
    public function index() {
        $tasks = Task::all();
        return view('pages.task', compact('tasks'));
    }
}
```

Now you must create a ‘task.blade.php’ file inside ‘resources/views/pages’ folder. I won’t repeat the HTML part. You need to understand one important thing. In your TaskController you catch all data from database in a variable called ‘\$tasks’. The previous ‘json’ output shows that it’s actually an array. What you need is a ‘foreach’ loop to retrieve them inside your ‘task.blade.php’ page.

```
@foreach ($tasks as $task)
    <h3>
        {{ $task->title }}
    </h3>
    <div class='h2'>
        {{ $task->body }}
    </div>
    Reminder :
    <div class='h1'>
        {{ $task->reminder }}
    </div>
@endforeach
```

For each ‘\$tasks’ as ‘\$task’ we can directly get the data. The workflow is completed.

10.1 Summary

It started from building up an environment file where we mentioned the default database as MySQL and named the database as ‘testdb’. Next we did the database migrations and using ‘tinker’ added and updated a few tasks. Further, we moved ahead and created an eloquent model class ‘Task’ since our database table name was ‘tasks’. Next we registered the route and created a TaskController class and through it we passed the data using ‘Task’ model to our view page ‘task.blade.php’.

10.1.1 Our Next Challenge

For a small text-based database application, SQLite is fine. It’s fast and file based. Let us build a database-driven application that will create, retrieve, update, and delete ‘reviews’. This time we’ll not use ‘tinker’ for database operations. Instead we’ll learn about forms and build separate pages for those jobs.

The ‘review’ page of a certain user, John Doe for example, will have a home page that will list the titles. If someone clicks that title she can read that review in a separate page. At the same time John can manage the whole database through a dashboard where authentication will be needed. Through that dashboard he will manage the ‘CRUD’. This time we use our Laravel/homestead box and name this application as ‘review.app’, so that if you type <http://review.app> you can see John’s reviews. And you can also insert, update, and delete data; it will be a total CRUD application.

*Let us stomp our feet on the beach!
It’s fun...*

CHAPTER 11



SQLite Is a Breeze!

We'd like to make this application entirely based on SQLite database. Normally, for a big application people opt for MySQL or PostgreSQL as it can tackle 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 a hundred thousand visitors. So we can feel free to use it for our CRUD application. Especially for Laravel, SQLite is a breeze to use. In a minute you'll understand why I sound so confident.

To use SQLite in Laravel you need to change the default database setup. Two lines need to be changed.

```
//Code/test/blog/config/databse.php
'default' => env('DB_CONNECTION', 'sqlite'),
```

In the second line, you need to mention the SQLite database file path.

```
//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' => '',
],
```

We are going to keep our SQLite file in the 'Code/test/blog/database/' folder. Many people go for the storage folder. Any one of them will work.

Secondly, we need to change the '.env' file. In the original file that comes with Laravel, the default database is mentioned as 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
```

Therefore, from now on, whatever database operation you'd do on your application will automatically be registered on SQLite database file that you create in the 'Code/test/blog/database' folder. Normally, people don't go for creating a new SQLite file in 'Code/test/blog/database' folder. They choose the 'Code/test/blog/storage/database.sqlite' file that comes with Laravel by default. In that case, in your database path 'Code/test/blog/config/database.php' you need to change the path.

We want to take a little break from usual path and create a 'database.sqlite' at 'Code/test/blog/database' folder. To do that you must go to the desired folder first.

```
cd Code/test/blog/database
```

Next we have to use 'touch' command to create the file.

```
touch database.sqlite
```

Now you're ready to make an awesome CRUD application using SQLite. And to do that, the first thing you have to do is fill up the forms with inputs. Through those inputs, you can insert or update data into your file-based SQLite database.

CHAPTER 12



Fiddly Feelings of Forms

To fill up the forms, you need to feel the forms first. Since Laravel 5.2.45 this feeling is a little bit different. It's become little bit tricky.

You need to install HTML form builder so that you can use Laravel form blade template; this is really important for security reasons. But, there is a catch. Since Laravel 5.2.45 the old trick will not work. You can no longer use `'composer require illuminate/html'` command to add form builder templates to your project. Instead, you have to issue this command:

```
sudo composer require laravelcollective/html
```

It'll take few minutes for composer to update your system. Next it'll search for 'providers' and 'aliases'. So you must open 'Code/test/blog/config/app.php' file and this line inside 'providers' array.

```
//Code/test/blog/config/app.php
'providers' => [

    /*
     * Laravel Framework Service Providers...
     */
    ...
    'Collective\Html\HtmlServiceProvider',
],
```

Next you must add two lines between the 'aliases' array.

```
//Code/test/blog/config/app.php
'aliases' => [

    ...
    'Form' => 'Collective\Html\FormFacade',
    'Html' => 'Collective\Html\HtmlFacade',

],
```

Now you're fully loaded to use the Laravel form template. In the next chapter we'll see extensive use of the forms.

CHAPTER 13



A CRUD Application

I don't want to elaborate this application because I believe you're smart enough to follow what will take place in a few minutes when you'll have a look at the subsequent codes.

The application we're going to make is very simple. John Doe is a user who wants to create a review page for his web site. He must have an index page. From that index page one can go the detail section of the reviews he has been keeping. To do that, he has a database migration file first.

First we create that file (consult the database migration chapter). Our file is named something like this: `2016_09_03_015541_create_reviews_table.php`.

The code is like this:

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

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

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



```

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

```

You have already chosen SQLite database file to keep your data. So the next part is pretty simple. Write down the Model, Views, and Controllers.

The model we use in this case, the old 'Task' model will not work any more. We need to create a new 'Review' model. Create it according to the trick you've learned. Issue the artisan command. It'll automatically be created.

The 'Review.php' file is created in your 'Code/test/blog/app' folder.

```

//Code/test/blog/app/ Review.php
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;

class Review extends Model
{
    protected $fillable = [
        'category',
        'title',
        'content',
        'rating'
    ];
}

```

Next you create a controller class. You have learned before how to create the controller class: 'TaskController.php' file.

```

//Code/test/blog/app/Http/Controllers/ReviewController.php
<?php
namespace App\Http\Controllers;

use App\Review;
use App\Http\Requests;
use Illuminate\Http\Request;

class TaskController extends Controller
{

    protected $reviews;
    protected $requests;
}

```

```

public function __construct(Review $reviews, Request $requests) {

    $this->reviews = $reviews;
    $this->requests = $requests;
}

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

public function show($id) {
    //return $tasks by id;
    $tasks = Review::findOrFail($id);
    return view('pages.show', compact('tasks'));
}

public function create() {
    return view('pages.create');
}

public function store(Request $request){
    //$input = Request::all();
    //return $input;
    //return $request->title;
    $review = new Review();
    $review->category = $request->category;
    $review->title = $request->title;
    $review->content = $request->content;
    $review->rating = $request->rating;
    $review->save();
    return "Success";
}

public function editreview() {
    $tasks = $this->reviews->get();
    //return all $tasks;
    return view('pages.editreview', compact('tasks'));
}

public function edit($id)
{
    $tasks = $this->reviews->find($id);
    return View('pages.edit', compact('tasks'));
}

public function update($id)
{
    $review = $this->reviews->find($id);
    $request = $this->requests;
    $review->category = $request->category;

```

```

        $review->title = $request->title;
        $review->content = $request->content;
        $review->rating = $request->rating;
        $review->save();
        return "Success";
    }
}

```

From the controller class you understand that there are a few new methods other than the 'index'. The index method shows the home page of John Doe's review page (Figure 13-1).

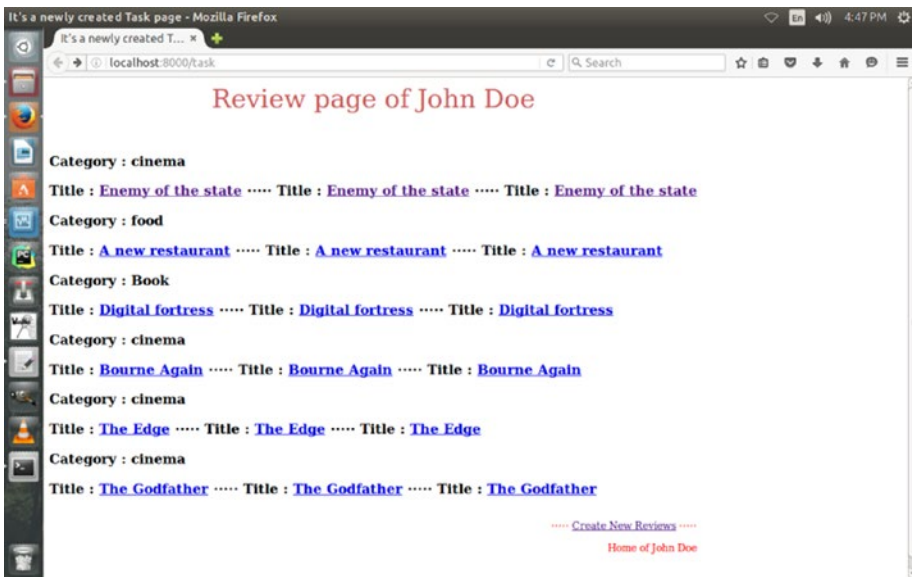


Figure 13-1. The review page of John Doe

The controller method is like this:

```

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

```

The code of 'index.blade.php' page in the views folder is a mere repetition of the previous codes we had used in our early application development. It extends a master layout and follows a few simple rules.

```

//resources/views/pages/index.blade.php
@extends('pages.master')
@section('head')

@stop
@section('title')
    It's a newly created Task page
@stop
@section('heading')
    Review page of John Doe
    <p></p><p></p><p></p> <p></p><p></p><p></p>
@stop
@section('content')
    <p></p><p></p><p></p> <p></p><p></p><p></p>
    @foreach ($tasks as $task)
        <h3>
            Category : {{ $task->category }}
        </h3>
        <h3>
            Title : <a href="/task/{{ $task->id }}">{{ $task->title }}</a>
            &sdot;&sdot;&sdot;&sdot;&sdot;
            Title : <a href="{{ action('TaskController@show', [$task->id]) }}">{{
            $task->title }}</a>
            &sdot;&sdot;&sdot;&sdot;&sdot;
            Title : <a href="{{ url('/task', $task->id) }}">{{ $task->title }}</a>
        </h3>
    @endforeach
@stop
@section('footer')
    <p></p>
    <p></p><p></p><p></p><p></p><p></p><p></p><p></p>
    <p></p><p></p><p></p><p></p>
    &sdot;&sdot;&sdot;&sdot;&sdot;
    <a href="{{ url('/create') }}">Create New Reviews</a>
    &sdot;&sdot;&sdot;&sdot;&sdot;
    <p></p>
    <p></p><p></p><p></p><p></p><p></p><p></p><p></p>
    <p></p><p></p><p></p><p></p>
    Home of John Doe
@stop

```

The required route for this page will be quite simple. We assume reviewing is our new task so we keep it as a simple task.

```

//app/Http/routes.php
Route::get('task', 'TaskController@index');

```

Once this index page of John Doe’s review has shown up, the visitor will be happy to click the titles and read the reviews. For that we’ve used the special ‘url’ function. Watch this part of the ‘index.blade.php’ page.

```
<h3>
    Title : <a href="/task/{{ $task->id }}">{{ $task->title }}</a>
    &sdot;&sdot;&sdot;&sdot;&sdot;
    Title : <a href="{{ action('TaskController@show', [$task->id]) }}">{{
    $task->title }}</a>
    &sdot;&sdot;&sdot;&sdot;&sdot;
    Title : <a href="{{ url('/task', $task->id) }}">{{ $task->title }}</a>
```

We’ve used three methods just to show you the various flexible methods. In the real world you must use only one. If you ask my preference, I’ll vote for any one of them!

The clickable links will take the visitors to the ‘show.blade.php’ page in the views folder (Figure 13-2).

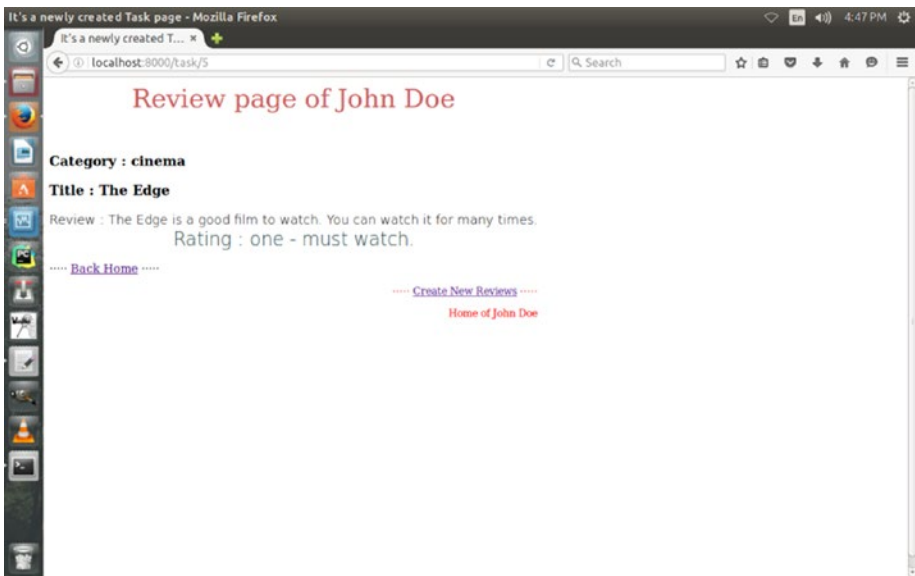


Figure 13-2. The detail of the review page

It’s fairly simple in our ‘TaskController’ part. Watch this code segment.

```
//TaskConreoller.php
public function index() {
    $tasks = $this->reviews->all();
    return view('pages.index', compact('tasks'));
}
public function show($id) {
```

```

    $tasks = Review::findOrFail($id);
    return view('pages.show', compact('tasks'));
}

```

The code of the 'show.blade.php' is a little bit tricky. But it needs no explanation at this stage.

```

//show.blade.php
@extends('pages.master')
@section('head')
@stop
@section('title')
    It's a newly created Task page
@stop
@section('heading')
    Review page of John Doe
@stop
@section('content')
    <h3>
        Category : {{ $tasks->category }}
    </h3>
    <h3>
        Title : {{ $tasks->title }}
    </h3>
    <div class='h2'>
        Review : {{ $tasks->content }}
    </div>
    <div class='h1'>
        Rating : {{ $tasks->rating }}
    </div>
<p></p>
<a href="{{ url('/task') }}">Back Home</a>
&sdot;&sdot;&sdot;&sdot;&sdot;
@stop
@section('footer')
<p></p>
<a href="{{ url('/create') }}">Create New Reviews</a>
    Home of John Doe
@stop

```

Now we can route this page through the 'routes.php' file like this:

```

//routes.php
Route::get('task/{id}', 'TaskController@show');

```

Now the important part of the application comes up slowly. First comes the creation part. John Doe wants to create or add reviews to his pages. He will add categories, title, content and rating, one after another (Figure 13-3).

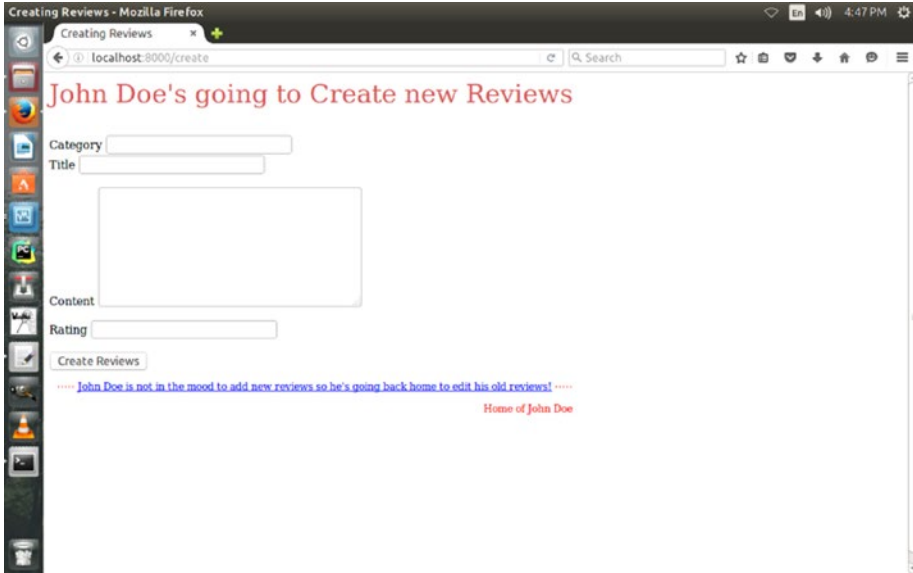


Figure 13-3. Create review page

The process is not very difficult. In a simple php application it'd have taken probably a few hundred lines of code to make such an application. But Laravel makes life much simpler.

In the 'TaskController' class the segment seeks your attention. We've written this part with two methods consecutively.

```
//TaskController.php
public function create() {
    return view('pages.create');
}
public function store(Request $request){
    //$input = Request::all();
    //return $input;
    //return $request->title;
    $review = new Review();
    $review->category = $request->category;
    $review->title = $request->title;
    $review->content = $request->content;
    $review->rating = $request->rating;
    $review->save();
    return "Success";
}
```

Next we must have a blade template view page where we can use the forms. The code of 'create.blade.php' is like the following.

```

//create.blade.php
@extends('pages.master')
@section('head')

@stop
@section('title')
    Creating Reviews
@stop
@section('heading')
    John Doe's going to Create new Reviews
@stop
@section('content')

<div class="formgroup">
    {!! Form::open(['url' => 'create/task']) !!}
    <div class="formgroup">
        {!! Form::label('category', 'Category') !!}
        {!! Form::text('category', null, ['class' => 'formcontrol',
        'required']) !!}
    </div>
    <div class="formgroup">
        {!! Form::label('title', 'Title') !!}
        {!! Form::text('title', null, ['class' => 'formcontrol',
        'required']) !!}
    </div>
</p>

</p>
<div class="formgroup">
    {!! Form::label('content', 'Content') !!} {!!
    Form::textarea('content', null, ['class' => 'formcontrol',
    'required']) !!}
</div><p></p>
<div class="formgroup">
    {!! Form::label('rating', 'Rating') !!}
    {!! Form::text('rating', null, ['class' => 'formcontrol',
    'required']) !!}
</div>
<p>
<p></p>
<p></p>
{!! Form::submit("Create Reviews") !!}
{!! Form::close() !!}
</div>
@stop
@section('footer')
<a href="{{ url('/editreview') }}">John Doe is not in the mood to add
    new reviews so he's going back home to edit his old reviews!</a>

```



```
Home of John Doe
@stop
```

And to register the required route, we have to write this piece of code in our ‘routes.php’ file.

```
//routes.php
Route::get('create', 'TaskController@create');
Route::post('create/task', 'TaskController@store');
```

Finally, the most tricky part of our CRUD application is about to come. It sounds tricky but Laravel makes it much easier with the concept of model binding. Since the model ‘Review’ essentially plays the main role behind the scene, the ‘edit’ section is basically dealt by a special method.

First thing is our controller part.

```
//TaskController.php
public function edit($id)
{
    $tasks = $this->reviews->find($id);
    return View('pages.edit', compact('tasks'));
}
public function update($id)
{
    $review = $this->reviews->find($id);
    $request = $this->request;
    $review->category = $request->category;
    $review->title = $request->title;
    $review->content = $request->content;
    $review->rating = $request->rating;
    $review->save();
    return "Success";
}
```

As you see, we needed two methods. The first is the ‘edit’ method, which takes us to the ‘edit.blade.php’ page. The second part is obviously the ‘update’ method, which does the work behind. But at the end of the day, it’s the model binding that does the real trick behind the scene.

First comes the ‘editreview’ page. If John Doe wants to edit his reviews, he will type this URL into his browser: <http://test.app/editreview>.

We’re going to see the code in our ‘editreview.blade.php’ page.

```
//editreview.blade.php
@extends('pages.master')
@section('head')

@stop
@section('title')
```

```

It's a newly created Task page
@stop
@section('heading')
John Doe's going to edit Review page
@stop
@section('content')
    @foreach ($tasks as $task)
        <h3>
            Category : {{ $task->category }}
        </h3>
        <h3>
            Title : <a href="{{ url('/edit', $task->id) }}">{{ $task->title }}</a>
        </h3>
    @endforeach

@stop

@section('footer')

<a href="{{ url('/create') }}">Create New Reviews</a>

    Home of John Doe
@stop

```

There's nothing very special on this page. He clicks a link and reaches the 'edit' page. The code of 'edit.blade.php' is tricky on one line. Watch out for this! (See Figure 13-4.)

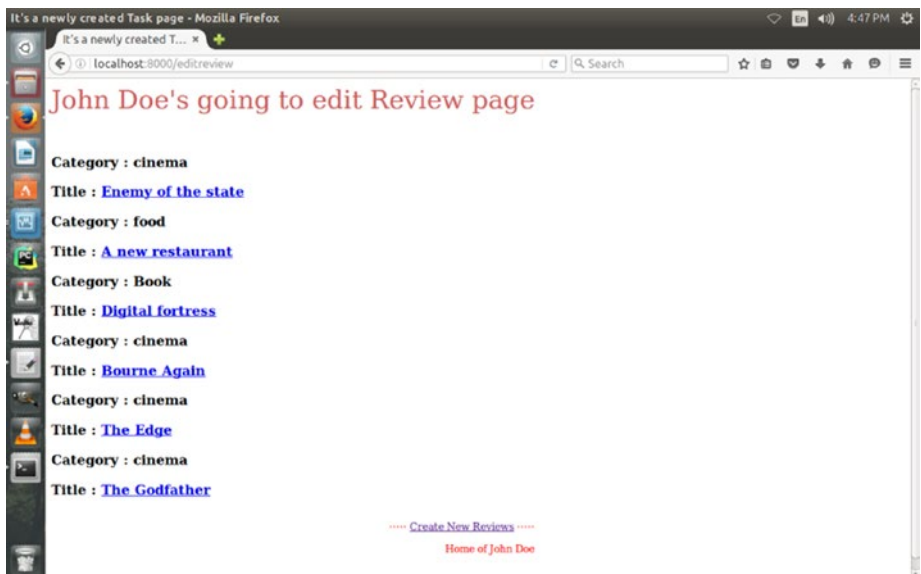


Figure 13-4. Edit page of John Doe's review

The edit page where the actual editing will take place will be a little different.

```
//edit.blade.php
@extends('pages.master')
@section('head')

@stop
@section('title')
    It's a newly created Task page
@stop
@section('heading')
    John Doe's going to edit Review page
@stop
@section('content')

<div class="formgroup">
{!! Form::model($tasks, ["url" => "update/$tasks->id", "method" => "PATCH"])
!!}
    <ul>
        <li>
            {{ Form::label('category', 'Category:') }}
            {{ Form::text('category') }}
        </li>
        <li>
            {{ Form::label('title', 'Title:') }}
            {{ Form::text('title') }}
        </li>
        <li>
            {{ Form::label('content', 'Content:') }}
            {{ Form::textarea('content') }}
        </li>
        <li>
            {{ Form::label('rating', 'Rating:') }}
            {{ Form::text('rating') }}
        </li>
        <li>
            {{ Form::submit('Update', array('class' => 'btn btn-info')) }}
        </li>
    </ul>
{!! Form::close() !!}

</div>

@stop

@section('footer')

    <a href="{{ url('/create') }}">Create New Reviews</a>
```

Home of John Doe
@stop

This part is tricky:

```
{!! Form::model($tasks, ["url" => "update/$tasks->id", "method" => "PATCH"])
!!}
```

The subsequent route will be registered in the ' routes.php' file.

```
//routes.php
Route::get('editreview', 'TaskController@editreview');
Route::get('edit/{id}', 'TaskController@edit');
Route::patch('update/{id}', 'TaskController@update');
```

The edit page of the John Doe's review looks like this when he clicks the relevant title to edit them (Figure 13-5).

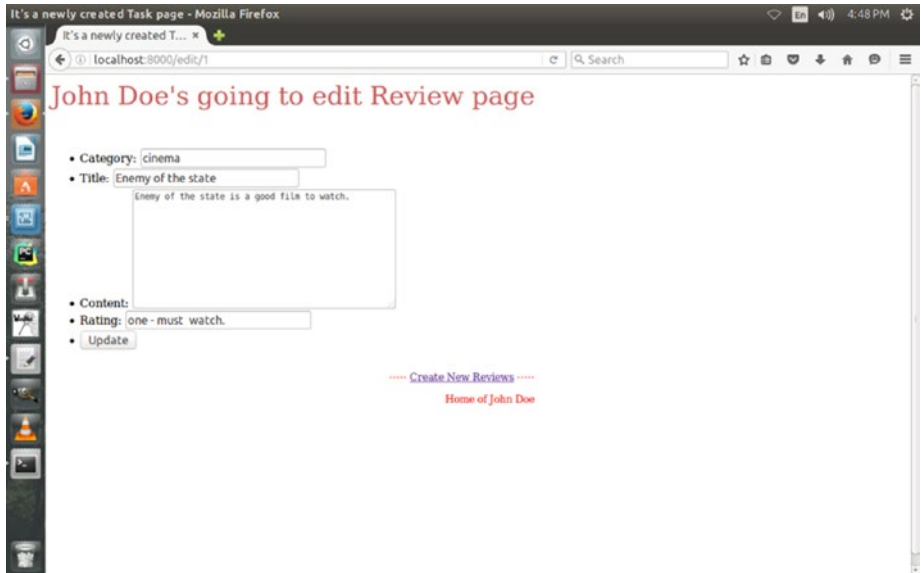


Figure 13-5. The editing page of John Doe's review

Our application is complete in a simple few lines of code. You did not need to write a hundred thousand of lines of code to accomplish this complex task.

The index page shows the titles of the reviews. When somebody clicks the titles, she can reach the review page. The 'show' method in the 'TaskController.php' has taken her to the detail of the reviews. When John Doe wants to add reviews it's quite easy. He adds it just by filling up his forms. Next comes the easiest part: the editing section. He clicks the title and reaches the edit page.

CHAPTER 14



Authentication and Authorization

Laravel 5.3 makes Authentication extremely simple, and if you're familiar with .NET Framework, it will remind you of the ASP.NET login and logout process.

In fact you need not have to do anything at all. Everything is ready at your doorstep; all you need is just plug in and switch on, and the Authentication process will work like a breeze. People coming from Laravel 3.2 or 4.2 backgrounds, find it unbelievable at times. In fact you are not supposed to write long lines of code. It is so easy and handy. It works just out of the box because everything has been configured beforehand and you can go through the code of 'config/app.php'. When you install Laravel 5, you see two controllers have already been set up in 'app/Http/Controllers/Auth/' folder. These two controllers are 'AuthController' and 'PasswordController'. Let us see how 'AuthController' works. It is used for new user registration and login purposes. And through 'PasswordController' a user can reset her password. Each of these controllers uses Traits to have their necessary methods. So you initially you should not change these controllers, unless you are in a very special situation where you would like to customize your registration and login process. Let us see first what lies inside this 'AuthController' Controller, so that we can understand how it works out of the box.

```
<?php

namespace App\Http\Controllers\Auth;

use Illuminate\Support\Facades\Auth;

use App\User;

use Validator;

use App\Http\Controllers\Controller;

use Illuminate\Foundation\Auth\ThrottlesLogins;

use
```

```

Illuminate\Foundation\Auth\AuthenticatesAndRegistersUsers;

use App\Http\Requests;
use Illuminate\Http\Request;

class AuthController extends Controller
{
    /**
     | Registration & Login Controller-----
     -----| This controller handles the registration of new
           | users, as well as the
     | authentication of existing users. By default, this controller uses
     | a simple trait to add these behaviors. Why don't you explore it?
    */
    use AuthenticatesAndRegistersUsers, ThrottlesLogins;
    protected $redirectPath = '/dashboard';
    protected $loginPath = 'auth/login';

    /**
     * Create a new authentication controller instance.
     *
     * @return void
     */
    public function __construct()
    {
        $this->middleware('guest', ['except' =>
'getLogout']);
    }
}

```

```

/**
 * Get a validator for an incoming registration request.
 *
 * @param array $data
 * @return
 */
\Illuminate\Contracts\Validation\Validator
 */
protected function validator(array $data)
{
    return Validator::make($data, [
        'name' => 'required|max:255',
        'email' => 'required|email|max:255| unique:users',
        'password' => 'required|confirmed| min:6',
    ]);
}
/**
 * Create a new user instance after a valid registration.
 *
 * @param array $data
 * @return User
 */
protected function create(array $data)
{
    return User::create([

```

```

        'name' => $data['name'],
        'email' => $data['email'],
        'password' =>
bcrypt($data['password']),
    ]);
}
}

```

When you create an instance of Authentication, it automatically calls back the ‘middleware’ property, and in your route you can set it so that if want to protect any page from unsigned visitor or guest you just mention it in your route.

```

public function __construct()

{
    $this->middleware('guest', ['except' =>
'getLogout']);
}

```

The validator() method at the same time works so that when someone registers he is guided properly to maintain the rule.

```

protected function validator(array $data) {
    return Validator::make($data, [
        'name' => 'required|max:255',
        'email' => 'required|email|max:255|
unique:users',
        'password' => 'required|confirmed|min:6',
    ]);
}

```

So in a nutshell the ‘AuthController’ is shipped with all the guns to fire up in a crisis. Now let us start building up a nice Authentication process through which we can register and log in. And at the same time we will see how we can use ‘middleware’ property to

guard our users' dashboards. To start with you need to create a 'users' table. You are supplied with a default user migration file in your 'database/migration' file. Let us set it up first like this:

```
public function up()
{
    Schema::create('users', function
(Blueprint $table) {
        $table->increments('id');
        $table->string('name');
        $table->string('email')->unique();
        $table->string('password', 60);
        $table->rememberToken();
        $table->timestamps();
    });
}
```

You see I have set up the columns. Now issue the following command:

```
php artisan migrate
```

I have set up a database 'MyApp' in the '.env' file beforehand so the 'users' have been added to that database now. Now we add four view pages to our 'views' folder. These pages are 'login.blade.php', 'register.blade.php', 'dashboard.blade.php', and 'home.blade.php'.

```
//codes of 'login.blade.php' starting
```

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

```
@section('title')
```

```
    Login Page
```

```
@stop
```

```
@section('content')
```

```

<h1>
    Sign In
</h1>
@foreach ($errors->all() as $error)
<li>
    {{ $error }}
</li>
@endforeach
<div class="formgroup">

    {!! Form::open(["url" => "auth/login",
"method" => "POST"]) !!}
<div class="formgroup">
{!! Form::label('Name : ') !!} {!!
Form::text('name')!!}
</div>
<p></p>
<div class="formgroup">
{!! Form::label('Email : ') !!} {!!
Form::email('email')!!}
</div>
<p></p>
<div class="formgroup">
{!! Form::label('Password : ') !!} {!!

```

```

Form::password('password')!!}
</div>
<p></p>
<p></p><p></p>
{!! Form::submit('Sign In') !!}
{!! Form::close() !!}
</div>
@stop
@section('footer')
    Users Registration page
@stop
//code ended
    Now let us create register page.
//code of 'dashbaord.blade.php' starting
@extends('auth.master')
@section('title')
Registration Page
@stop
@section('content')
<h1>
    Register
</h1>
@foreach ($errors->all() as $error)
<li>

```

```

    {{ $error }}
</li>
@endforeach
<div class="formgroup">
    {!! Form::open(["url" => "auth/register",
"method" => "POST"]) !!}
<div class="formgroup">
    {!! Form::label('Name : ') !!} {!!
Form::text('name')!!}
</div>
<p></p>
<div class="formgroup">
    {!! Form::label('Email : ') !!} {!!
Form::email('email')!!}
</div>
<p></p>
<div class="formgroup">
    {!! Form::label('Password : ') !!} {!!
Form::password('password')!!}
</div>
<p></p>
<div class="formgroup">
    {!! Form::label('Password Confirmation : ') !!}
    {!! Form::password('password_confirmation')!!}
</div>

```

```

<p></p><p></p>
{!! Form::submit('Register') !!}
{!! Form::close() !!}
</div>

@stop

@section('footer')
    Registration Page
@stop

//code ended

```

For brevity we will skip the codes for dashboard and home. Everything is the same except for one logout option. That is like this:

```

<div class="formgroup">
    {!! Form::open(["url" => "auth/logout",
"method" => "POST"]) !!}
<p></p><p></p>
{!! Form::submit('Sign Out') !!}
{!! Form::close() !!}

```

Now finally we add these methods 'routes.php'

```

//codes of 'routes.php' starting
// Authentication routes
Route::get('auth/login',
'Auth\AuthController@login');
Route::post('auth/login',
'Auth\AuthController@postLogin');
Route::get('auth/logout',

```

```

'Auth\AuthController@getLogout');
    Route::post('auth/logout',
'Auth\AuthController@getLogout');
    // Registration routes
    Route::get('auth/register',
'Auth\AuthController@getRegister');
    Route::post('auth/register',
'Auth\AuthController@postRegister');
//how to protect your users page
Route::group(['middleware' => 'auth'], function ()
{
    Route::get('home', function () {
        return view('auth.home');
    });
    Route::get('dashboard', function () {
        return view('auth.dashboard');
    });
    //it works
    //Route::get('test', 'MyController@test');
});
//code ended

```

Now your register page automatically does the server-side validation. You see the 'register.blade.php' page `$errors->all()` method. It does the trick. If a user fills up all the fields properly and presses the register button, he will be directed to the home page automatically. And if he logs in he will be redirected to the dashboard. There he can sign out. So we can conclude that Authentication in Laravel 5 is just like a cakewalk. All you need to do is set the users table, create a few view pages, and add the necessary routes.

CHAPTER 15



More About Validation

Okay, there is nothing new in the word validation. Even if you are a beginner you already know what the term validation means. Very simple. Sending blank forms might cause major problem for the server and as a developer you would also suffer from that. Another big problem is the user itself. The rule of thumb is not to trust the user. And that is true. You want user's e-mail and some text is being supplied instead. How will tackle this menace? The answer is validation. And Laravel has made it really easy. Not only easy, but with so many choices, it really becomes something where the possibilities are endless. Let us consider a form like before:

```
{{ Form::open(array('url' => '/')) }}  
  
{{ Form::label('username', 'Username') }}  
  
{{ Form::text('username') }}  
  
{{ Form::label('email', 'EMail Address') }}  
  
{{ Form::email('email', 'me@sanjib.me') }}  
  
{{ Password field. }}  
  
{{ Form::label('password', 'Password') }}  
  
{{ Form::password('password') }}  
  
{{ Password confirmation field. }}  
  
{{ Form::label('passwordConfirmation', 'Password  
confirmation') }}  
  
{{ Form::password('passwordConfirmation') }}  
  
{{ Form::open(array('url' => '/')) }}
```

```

{{ Form::hidden('hagu', 'mutu') }}

{{ Form::open(array(

'url'

=> 'test/test',

'files' => true

)) }}

{{ Form::label('avatar', 'Avatar') }}

{{ Form::file('avatar') }}

{{ Form::submit('Submit') }}

{{ Form::close() }}

//end of code

```

The form is simple enough with some tricks included. But it targets the main opening page that can be changed to something like 'registration.blade.php' or anything you like. And by default it uses 'POST' request verb. Now, how can we validate this form so that no one sends a blank request, causing severe problems for our application and burdening our server? Our first priority should be the form fields that should not have been left blank. A user must fill in the forms. Our application would look into that matter internally. Remember, the greatness of Laravel is that it usually looks after this mechanism fantastically. You need not worry about it at all. As you see, we can set more rules for our users here. For example, in username, there must be some alphanumeric value. We should force our user to follow the rule, otherwise it will not be maintained. As I say, the possibilities are endless; there are many more options. The rule is very simple. And Laravel will take the main responsibilities so you need not worry about it at all.

It goes something like this:

```

Route::post('/', function()

{

$postdata = Input::all();

//we are going to handle the form

});

//end of code

```


And the next step would be pretty simple:

```
Route::post('/', function()
{
$postdata = Input::all();
$check = array(
'username' => 'alpha_num'
);
});
//end of code
```

As you see, we set our users follow a rule which says that you should provide some alphanumeric value. Okay, I hope this is enough to provide a sample of how it works. Before embarking on a new journey to Laravel validation, we would like to see if we could have done some validation on our own if we were not given a superpower like Laravel! It will help us to clear up the concepts.

We're going to test almost same form making it a little shorter for brevity. We have three files: 'form.php', 'validate.php', and 'action.php'. In the first file, 'form.php', the form will show up and is ready for checking two fields: username and e-mail. We force our users to obey some rules, like no blank submissions, e-mail must look like e-mail, et cetera.

```
//first the form view page:
<h1>
    The Crazy ByCycle Club
</h1>
<h2>
    Please Fill up your Credentials for the
crazy tour!
</h2>
<form method="POST" action="action.php"
acceptcharset="UTF8">
```

```

        <input name="_token" type="hidden"
value="EMnZhqPbryk7oVPcrjwxuTrlHto">
<label for="username">Your Name</label>
<p>
<input name="username" type="text" value="Your      name" id="username">
<p>
<label for="email">Your Email</label>
<p>
<input name="email" type="text" value="your email"
id="email">
<p>
<input type="submit" value="Register">
/end of code

```

It's a very simple form page where you are about to register your username and e-mail. Let us see how we can make it validate it in our 'validate.php' class. The validation class code goes like this:

```

//code starts
<?php
/*
* .validate class checks two things in user name
field
* 1) user name should not be left blank
* 2) user name must be between 3 to 8 characters
* Secondly it checks the email field
* email should have a @ sign

```

```

*/
trait ValidateTrait {
    public function checked() {
        return "Okay, Your Data has been saved";
    }
    public function unchecked() {
        return "You have not provided proper data.";
    }
}

interface ValidateInterface {
    public function make($value);
}

class Validate implements ValidateInterface {
    use ValidateTrait;

    public $_value = array();

    public function make($value) {
        $this->_value = $value;

        if (strlen($value) === 0 || strlen($value)
< 3 || strlen($value) > 8){
            return FALSE;
        }

        elseif(is_string($value) && trim($value)
=== ''){
            return FALSE;
        }
    }
}

```

```

        elseif (is_array($value)) {
            return FALSE;
        }

        elseif (preg_match("/\@.\./i", $value)) {
            return FALSE;
        }

        return TRUE;
    }
}

//end of code

```

Next in 'action.php' we will validate this formdata. The code is like this:

```

//code starts

<?php
/*
* validating form
*/

require 'validate.php';

if ($_POST['_token'] === "EMnZhqPbryk7oVPcrjwxuTrlHto"){
    $value = [$_POST['username'],
$_POST['email']];

    $validate = new Validate();

    if ($validate>make($value[0]) && $validate
>make($value[1])){
        echo $validate>checked();
    }
}

```

```

    }
    else {
        echo $validate>unchecked();
    }
}
else {
    echo 'You Crackers! Go back!';
}
//end of code

```

We have passed a hidden token and matched it with our formdata so that it cannot be sent by an outsider. But it could be written in a better way. For brevity we have caught data directly but accessing super global array directly is not a good practice. It can be done like this:

```

//code starts
<?php
/*
 * validating form
 */
require 'validate.php';
$token = filter_input(INPUT_POST, '_token',
FILTER_SANITIZE_FULL_SPECIAL_CHARS);
$user = filter_input(INPUT_POST, 'username',
FILTER_SANITIZE_FULL_SPECIAL_CHARS);
$email = filter_input(INPUT_POST, 'email',
FILTER_SANITIZE_EMAIL);
if ($token === "EMnZhqPbryk7oVPcrjwxuTrlHto"){

```

```

    $value = [$user, $email];

    $validate = new Validate();

    if ($validate->make($value[0]) &&
        $validate->make($value[1])){

        echo $validate->checked();

    }

    else {

        echo $validate->unchecked();

    }

}

else {

    echo 'You Crackers! Go back!';

}

//end of code

```

If lines of coding and making things come together do matter to you, you will prefer to delegate your task to Laravel. The creator Taylor Otwell and the Laravel chore team look after all the possibilities and kept close to the principle that no stone should be unturned. So everything goes into its proper place and you are rendered with every possible option to optimize your application. Let us go to our old bicycle form and see how we can validate it, forcing our users to go through many tasks that are available in Laravel. We have ended here before:

```

Route::post('/', function()
{
$postdata = Input::all();

$check = array(
'username' => 'alpha_num'
);
});

//end of code

```

Now we know what it tells. But how we can validate this? Is there any validate class? Yes, there is! Let us find out how it can be used. To do that we need to set up our validate object that will check whether the rule is strictly followed or not! The code goes like this:

```
//code starts

Route::post('/', function()
{
// Fetch all request data.
$postdata = Input::all();
// Build the validation constraint set.
$check = array(
'username' => 'alpha_num'
);
// Creating instance that should be passed.
$validatingData = Validator::make($postdata,
$check);
});

//end of code
```

This code is not complete, as you have guessed, as our validation instance should be passed through a rigorous test. We will do that shortly. This 'postdata' can be anything as you define it accordingly. No matter. Finally, the full text of the code goes like this:

```
//code starts

Route::post('/', function()
{
// Fetch all request data.
$postdata = Input::all();
// Build the validation constraint set.
```

```

$check = array(
'username' => 'alpha_num'
);

$validingData = Validator::make($postdata,
$check);

if ($validingData->passes()) {
// when it passes you can insert, update your //database. If it did not pass
you can return redirect to a //failure page also return 'Data was saved.'
}

return Redirect::to('/success');

});

//end of code

```

Now you have got the main arsenal, and how you fire from it is up to you. There are obviously many other options available alongside the alphanumeric characters. In this form, if we put a name like 'sanjib65' it will pass throughout and the data will be saved. But instead of alphabetical or numerical value we put some strange characters like '@!!&*', it will fail. In such case, we can use our code this way:

```

//code starting

Route::post('/', function()
{
// Fetch all request data.
$postdata = Input::all();
// Build the validation constraint set.

$check = array(
'username' => 'alpha_num'
);

$validingData = Validator::make($postdata,

```



```

$check);

if ($validatingData>fails()) {
    // Normally we would do something with the data.

    return Redirect::to('/failure');
}

return 'Data saved.';
});

//end of code

```

It is fairly simple, so no explanation is needed. Let us see how we can add many more features to our '\$check' array. Suppose we want our username to be a minimum of six characters long. What we can do? Very simple. Laravel makes it writing like a explicit description: min:6. This 'min:6' can be changed to 'min:4' or 'min:5' or any length you need to adopt. The code is almost the same, with a little change, as you supposed.

```

//code starting

Route::post('/registration', function()
{
    // Fetch all request data.

    $postdata = Input::all();

    // Build the validation constraint set.

    $check = array(
        'username' => 'min:6'
    );

    $validatingData = Validator::make($postdata, $check);

    if ($validatingData>passes()) {
        // Normally we would do something with the data. return 'Data was saved.';
    }
}

```

```
return Redirect::to('/success');
});
//end of code
```

Instead of 'alpha_num' you write 'min:6' and that is it. But wait! There is more. Suppose you can add both features to your username. What can you do? Laravel's greatest feat is that it makes things so simple and elegant that you cannot imagine before using it how it is being rendered. The full code goes like this:

```
//code starting
Route::post('/registration', function()
{
// Fetch all request data.
$postdata = Input::all();
// Build the validation constraint set.
$check = array(
'username' => 'alpha_num|min:6'
);
$validatingData = Validator::make($postdata,
$check);
if ($validatingData->passes()) {
// Normally we would do something with the data.
return 'Data was saved.';
}
return Redirect::to('/success');
});
//end of code
```

Now you already know that Laravel ships with a lot of validation facilities. And all the tricks have emerged from the Validation class. You are free to check out the chore what are all the methods that are being used. Besides, you can go throughout the documentation that nicely explains the rules, one after another. Here I am trying to give you a glimpse of what you can do with this Validation class. Let us check this code again:

```
//code starting

$validatingData = Validator::make(
    array('name' => 'Sanjib'),
    array('name' => 'required|min:6')
);

//end of code
```

Validator class has a make method which takes the first argument as the data that are to be checked. And the second argument shows the requirements. You can either use the 'pipe' character or you can use array to separate elements like this:

```
//code starting

$validatingData = Validator::make(
    array('name' => 'Sanjib'),
    array('name' => array('required', 'min:6'))
);

//end of code
```

You can also use multiple fields:

```
//code starting

$validatingData = Validator::make(
    array(
        'name' => 'Sanjib',
        'password' => 'mypassword',
        'email' => 'me@sanjib.me'
```

```

),
array(
  'name' => 'required',
  'password' => 'required|min:8',
  'email' => 'required|email|unique:users'
)
);
//end of code

```

You have already seen how we can use ‘passes’ and ‘fails’ methodology to save or reject formdata. We can also use ‘message’ method to get the error messages. It is very simple:

```

//code starting
$validatingData = Validator::make(
  array('name' => 'Sanjib'),
  array('name' => 'required|min:6')
);
$validatingData = Validator::make(
  array(
    'name' => 'Sanjib',
    'password' => 'mypassword',
    'email' => 'me@sanjib.me'
  ),
  array(
    'name' => 'required',
    'password' => 'required|min:8',
    'email' => 'required|email|unique:users'

```

```

)
);
$messages = $validator->messages();

//end of code

```

There is another option to retrieve error messages through ‘failed’ method.

```
$failedMessage = $validator>failed();
```

By using this method, you can grab all the failed rules that your user did not properly obey while filling up the formdata. Now obviously, question is how we can retrieve the error messages or failed rules? When you call the ‘message’ method on a Validator instance, it usually comes up with an array. Obviously, you have set many parameters like this:

```

//code starting

array(
'name' => 'required',
'password' => 'required|min:8',
'email' => 'required|email|unique:users'
)

//end of code

```

Now you can get one by one – just like this:

```
echo $messages->first('email');
```

Or you can get it as a whole:

```

//code starting
foreach ($messages>get('email') as $message)
{
echo $message . "<br>";
}

//end of code

```

Suppose you have a code like this where you have already set plenty of validating rules and you want to grab all the error messages at one go. How you can do that?

```
//code starting

$validatingData = Validator::make(
array('name' => 'Sanjib'),
array('name' => 'required|min:6')
);

$validatingData = Validator::make(
array(
'name' => 'Sanjib',
'password' => 'mypassword',
'email' => 'sanjib12sinha@gmail.com'
),
array(
'name' => 'required',
'password' => 'required|min:8',
'email' => 'required|email|unique:users'
)
);

$errorMessages = $validator->messages();
foreach ($errorMessages>all() as $message)
{
echo $message . "<br>";
}

//end of code
```

Do you want your error messages in a nice formatted way? Laravel thinks about this also. You can write your whole code like this:

```
//code starting

$validatingData = Validator::make(
array('name' => 'Sanjib'),
array('name' => 'required|min:6')
);

$validatingData = Validator::make(
array(
'name' => 'Sanjib',
'password' => 'mypassword',
'email' => 'me@sanjib.me'
),
array(
'name' => 'required',
'password' => 'required|min:8',
'email' => 'required|email|unique:users'
)
);

$errorMessages = $validator->messages();
foreach ($errorMessages->all('<li>:message</li>')
as $message)
{
echo $message . "<br>";
}

//end of code
```

As you see, there are plenty of options. Taylor and his team have thought of almost all possibilities and tried to give you support in every possible way. Now let us imagine a more concrete example, where we have a 'register.blade.php' in our 'views' folder. We have a form like this:

```
//code starting

{{ Form::open(array('url' => '/')) }}

{{ Form::label('username', 'Username') }}

{{ Form::text('username') }}

{{ Form::label('email', 'EMail Address') }}

{{ Form::email('email', 'me@sanjib.me') }}

{{ Password field. }}

{{ Form::label('password', 'Password') }}

{{ Form::password('password') }}

{{ Password confirmation field. }}

{{ Form::label('password_confirmation', 'Password confirmation') }}

{{ Form::password('password_confirmation') }}

{{ Form::open(array('url' => '/')) }}

{{ Form::hidden('hagu', 'mutu') }}

{{ Form::open(array(

'url'

=> 'test/test',

'files' => true

)) }}

{{ Form::label('avatar', 'Avatar') }}

{{ Form::file('avatar') }}

{{ Form::submit('Submit') }}
```



```
{{ Form::close() }}
```

```
//end of code
```

Now in our route we can write something like this:

```
//code starting
```

```
Route::get('register', function()
```

```
{
```

```
return View::make('user.register');
```

```
});
```

```
Route::post('register', function()
```

```
{
```

```
$rules = array(...);
```

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

```
$rules);
```

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

```
{
```

```
return Redirect::to('register')
```

```
->withErrors($validator);
```

```
}
```

```
});
```

```
//end of code
```

Now when the validating rules fail and error messages generate, the `$error` variable is automatically defined to the session. The way Laravel tackles this problem is quite intelligent. You need not explicitly bind the error messages to your view in GET route. Laravel looks after this. How does it happen? Laravel always checks for the errors in session and binds them to the view. So if it is available, on every request `$error` variable is present in the view. Do you want anything more? :) So check for any field like this:

```
echo $errors->first('email');
```

Now it is time for checking for all the Validation rules. There are plenty and you can get the full list in the documentation. But let me get some of them so that you can get acquainted with them.

```
//code starting  
  
array(  
  'field' => 'accepted'  
);  
  
array(  
  'field' => 'alpha'  
);  
  
array(  
  'field' => 'url'  
);  
  
array(  
  'field' => 'size:8'  
);  
  
array(  
  'field' => 'alpha_dash'  
);  
  
array(  
  'field' => 'active_url'  
);  
  
array(  
  'field' => 'after:16/04/15'  
);
```

```
array(  
    'field' => 'before:09/02/15'  
);  
array(  
    'field' => 'unique:users,username'  
);  
array(  
    'field' => 'between:4,9'  
);  
array(  
    'field' => 'same:age'  
);  
array(  
    'field' => 'confirm'  
);  
array('field' => 'date');  
array('field' => 'required_without:age,height');  
array('field' => 'required_with:age,height');  
array('field' => 'required_if:username,sanjib');  
array('field' => 'required');  
array('field' => 'numeric');  
  
//end of code
```

I hope you would happily search for more validation rules in the Laravel documentation. You will get more and make your Validation rules more interesting. Here briefly I will have some discussion.

```
array(
    'field' => 'numeric'
);
```

Here you guess it correctly. The field must have a numeric value. A few of them are pretty interesting, like this one:

```
array(
    'field' => 'size:7'
);
```

Here ‘size’ stands for everything—numerical value, string, byte of an image—literally everything. As you know, size sometimes really matters for someone! Isn’t it very handy?

15.1 Conditional Rules

Sometimes in your input array a field may not be present but you want to check it. What can you do for that? Don’t worry! Laravel thinks about it already and ships with some solutions for your problem. For one field it is not a big problem as you can declare your variable like this:

```
$validatingData = Validator::make($data, array(
    'email' => 'sometimes|required|email',
));
```

But in a complex scenario it is quite different, as you have presumed. Suppose you are operating a hotel registration and booking site where there are more than one field for you to validate. For doing this, you need to check your client’s requests for rooms and number of people presentation.

You can check this conditional statement like this:

```
$validatingData = Validator::make($data, array(
    'email' => 'sometimes|required|email',
    'rooms' => 'required|numeric'
));
```

Now presumably you may want to ask why more than one room are requested. So in 'sometimes' method you can add that argument and hold it in your 'input' datatarget in the closure which is passed as third argument. There are more instances and for a detailed example the Laravel documentation will come handy. In this validation section, last but not the least is the Custom Validation part. Let us see how we can make some progress in it.

You can pass a custom message like this:

```
$yourmessage = array(
    'required' => 'The :attribute field is necessary.',
);
$validator = Validator::make($input, $rules, $yourmessage);
```

And you can pass an array of messages no matter how big you want it!

15.2 Custom Validation

So validation does not require much effort if you do know the tricks. The thing is, you need to know the classes and variables defined by default. But besides that you can also do some own customary validation rules. Laravel has a tremendous capacity to extend itself and even encourage you to write some awesome methods. So never feel shaky and always try to do justice to this expandable technique. Write your own staff and let the world know about it. Let us try to do some Custom Validation.

```
//custom validation
Validator::extend('customValidator',
function($field, $value, $parameters)
{ return $value == 'customValidator';
});
Route::get('/', function()
{
return View::make('form');
});
Route::post('/registrationPage', function()
```

```

{
// Getting all requested data.
$data = Input::all();

// Let us build the validation constraint set.
$rules = array(
'username'
=> 'customValidator',
);

// Creating a new validator instance.
$validingData = Validator::make($data, $rules);
if ($validingData->passes()) {
// here usually we do some database staff return 'Data was saved.';
}

return Redirect::to('/') ->withErrors($validingData);
});

//end of code

```

As you see, I just extended the Validator class with a method 'extend'. Basically what we wanted to do is build some custom validation rules. So in `Validator::extend()` method we need to pass those rules. We have used a Closure to do that. If we have a close look at the `extend()` method:

```

Validator::extend('customValidator',
function($field, $value, $parameters)
{
return $value == 'customValidator';
});

//end of code

```

We see that the first parameter is a nickname, nothing else. But the second parameter is the closure, which plays the main role.

It passes `$field`, `$value`, `$parameters`. The first parameter is the field (here it is 'username'), and the second one is the value of the field. But what about the third one which passes the parameters? It is an array of any parameters that have been passed to the validation rules. You can customize it throughout a custom class. No problem. And finally just to show the example I keep the following in our `app/routes.php` file:

```
Validator::extend('customValidator',
function($field, $value, $parameters)
{
return $value == 'customValidator';
});
```

But you can keep it in any file in your 'app'. And the same rule applies to the awesome custom class you are going to write. Suppose in 'app' you create a folder called 'validator' and keep your Custom Validation class inside it like this:

```
//custom validation class

class MyCustomValidation
{
public function customValidator($field, $value, $parameters)
{
return $value == 'customValidator';
}
}

//end of code
```

The question is: how you can use it inside your 'routes.php'? As you see inside the class, we return the nickname 'customValidator'. We can bind it to the `extend()` method just like this:

```
//starting of code

//custom validation
```

```
Validator::extend('customValidator',
MyCustomValidation@customValidator);
Route::get('/', function()
{
return View::make('form');
});
Route::post('/registrationPage', function()
{
// Getting all requested data.
$data = Input::all();
// Let us build the validation constraint set.
$rules = array(
'username'
=> 'customValidator',
);
// Creating a new validator instance.
$validatingData = Validator::make($data, $rules);
if ($validatingData->passes()) {
// here usually we do some database staff return 'Data was saved.';
}
return Redirect::to('/')
>withErrors($validatingData);
});
//end of code
```


The custom validation message is fairly simple and will not take much effort. In the existing custom validation rules you can do this. And no matter, it will not put you to a hard test. Let me reassure you that it is a painless and blissful coding experience. Let us see how we can use a custom validation message along with our 'MyCustomValidation' class.

```
//starting the code

//custom validation message

//custom validation

::extend('customValidator',

function($field, $value, $params)

{

return $value == 'customValidator';

});

Route::get('/', function()

{

return View::make('form');

});

Route::post('/registrationPage', function()

{

// Getting all requested data.

$data = Input::all();

// Let us build the validation constraint set.

$rules = array(

'username'

=> 'min:3',

);
```

```

//
$errorMessages = array('min' => 'Not less than three characters.');
```

// Creating a new validator instance.

```

$valdatingData = Validator::make($data, $rules, $errorMessages);
if ($valdatingData>passes()) {
// here usually we do some database staff return 'Data was saved.';
}
return Redirect::to('/') ->withErrors($valdatingData);
});
```

//custom validation class

//in app/validator/MyCustomValidation.php

```

class MyCustomValidation
{
public function customValidator($field, $value, $parameters)
{
return $value == 'customValidator';
}
}
//end of code
```

In the preceding code, this part is extremely important.

```

$errorMessages = array(
'min' => 'Not less than three characters.'
);
// Creating a new validator instance.
```

```

$validatingData = Validator::make($data, $rules, $errorMessages);

if ($validatingData->passes()) {

    // here usually we do some database stuff return 'Data was saved.';

}

return Redirect::to('/')->withErrors($validatingData);

});

//end of code

```

You see, in this `$validatingData = Validator::make($data, $rules, $errorMessages);` method, the third parameter is optional for passing error messages. It passes an array of messages. If you don't define it or customize it, it will pass a default message. If you have customary message it will override the default message. That is it. Here exactly the same thing happens.

15.3 How Form Validation Works

Form validation in Laravel 5 becomes extremely easy. You have seen that in the previous chapter. You don't have to validate your form fields separately. You just declare it in your Controller method and the scaffolding of Model and Views are ready to obey that rule. Let see how we can do it in the shortest way! We assume that we have a 'songs' table. I hope you didn't forget. I showed you before how to add new records through the form blade. Now let us check the 'SongsController' codes. Is there any validation method? No, so we need to create it first. Remember, Laravel's base controller class uses the 'ValidatesRequests' trait, which actually treats every HTTP request with a very powerful validation mechanism. So you need not worry about that. You need to add the `validate()` method only and once it has been added it will take care of every single HTTP request.

```

//code starting

public function store()

{

    $this->validate($this->request, [

        'title' => 'required|unique:songs|

max:255',

        'slug' => 'required||max:255'

```

```

    });

    $this->songs->title = $this>request
>title;

    $this->songs>slug = $this>request>slug;

    $this->songs->lyrics = $this>
request>lyrics;

    $this->songs->save();

    return 'Saved';

}

//code ended

```

And subsequently we added an `$errors->all()` method in our 'createsongs.blade.php' page like this:

```

@foreach ($errors->all() as $error)

<li>

    {{ $error }}

</li>

@endforeach

```

On your 'createsongs.blade.php' page you can catch your errors by this logic also:

```

@if (count($errors) > 0)

    <div class="alert alertdanger">

        <ul>

            @foreach ($errors->all() as

$error)

                <li>{{ $error }}</li>

            @endforeach

        </ul>

    </div>

```

```

        </ul>
    </div>

@endif

```

So there are many ways you can make your validation rules work. Please consult the documentation for the whole list of validation rules available. For an example, let us try a simple one. Change this line from that:

```
'title' => 'required|unique:songs|max:255',
```

to this:

```
'title' => 'required|unique:songs|max:255|min:2',
```

Now if you fill up the title field with one character, it immediately catches the error and shows it. The title must be at least two characters. And there are lot of options for you if want to make a custom Validation rule. You can use your artisan tool to create the custom validator class like this:

```
php artisan make:request StoreSongsPostRequest
```

Now the generated class will be stored in the 'app/Http/Requests' directory. Then you can add some custom rules like before:

```

public function myrules()
{
    return [
        'title' => 'required|unique:songs|
:255',
        'slug' => 'required||max:255|min:2',
        'lyrics' => 'required',
    ];
}

```

Now how can you get this custom Validation working? Very simple, you can either create the instance of newly created class in your construct or else use the store() method by typehinting the request object like this:

```

//code starting

public function store(StoreSongsPostRequest $request)
{
    $request->myrules($this->request, [
        'title' => 'required|unique:songs|
max:255|min:2',
        'slug' => 'required||max:255|min:2',
        'lyrics' => 'required',
    ]);

    $this->songs->title = $this->
request->title;

    $this->songs->slug = $this->request->slug;

    $this->songs->lyrics = $this->
request->lyrics;

    $this->songs->save();

    return 'Saved';
}

//code ended

```

At the end of this chapter, we can conclude that from Laravel 5 onward, the framework has made a developer's life much easier with a lot of tricks supplied. In fact, unless there is an extraordinary situation, you need not customize your Validation rules.



Eloquent Relations

Now we have learned to create tables through Migration and Tinker. We can also use Eloquent ORM to do the same. We also have learned to update or edit our database records. Suppose we have made three tables: 'users,' 'songs,' and 'singers.' Now in our 'songs' table we have two tablecells, as you have probably noticed. One is 'user_id' and the other is 'singer_id.' So presumably, a user inserts a song, and he has an ID. In my database I have inserted three users: first, second, and third. So they have user IDs like 1, 2, and 3, respectively.

I also inserted 10 songs, which of course have 10 respective song IDs. Finally I have five singers who have their songs in the 'songs' table. Remember, these singers might have one or two songs. Someone may not have any song at all. Besides, these singers also have a foreign key 'user_id'; that is, when a user inserts a singer's name in that table his ID is inserted along with it. So there is a relationship among these three tables, and these relationship can be explained in many formatted versions. But Eloquent makes the job of managing and working with these relationships easy, and supports several different types of relationships. They are as follows:

- One to One
- One to Many
- Many to Many
- Has Many Through
- Polymorphic Relations
- Many to Many Polymorphic Relations

Now we are going to explore these relationships one by one using Eloquent ORM in Laravel 4 also. After all, what is Eloquent ORM? Till now you have heard about MVC, that is ModelViewController methodology, but we did not say anything about the Model part.

Now, based on that theory, we assume that with each database table we have an associated Model. We can interact with the table much easily.

We have already seen how Laravel has made the whole database operation quite easy, and Eloquent is the new addition to it.

If you want a definition, Eloquent is the Active Record implementation for database operations. Now what is Active Record pattern? This is a technique that wraps database into objects. Suppose you have a 'user' table. You can present the data of that table as a class and every row as an object.

Now each database table has a corresponding Model that will do every operation that CRUD does. You know CRUD: create, retrieve, update, and delete. Now you can do them quite easily through Eloquent ORM. What is ORM? Object relation mapping. That is, you can have a defined set of relations between the corresponding tables.

As we have seen in our database query building process, a table may have lot of relations among other tables. Eloquent ORM has a built-in relationship. It makes our life much easier.

From Laravel 4.1 Eloquent becomes more powerful as it has attached a much-requested feature: polymorphic Many to Many relationship. Now you need not write a lot of code to establish a polymorphic Many to Many relationship between tables. Eloquent does that quite easily. We have 'users' table and we have a 'User' class in our Model like this:

```
//code starting

//models/User.php

<?php

use Illuminate\Auth\UserTrait;

use Illuminate\Auth\UserInterface;

use Illuminate\Auth\Reminders\RemindableTrait;

use Illuminate\Auth\Reminders\RemindableInterface;

class User extends Eloquent implements

UserInterface, RemindableInterface {

use UserTrait, RemindableTrait;

/**

* The database table used by the model.

*

* @var string

*/

protected $table = 'users';

/**

* The attributes excluded from the model's JSON form.
```



```

*
* @var array
*/
protected $hidden = array('password', 'remember_token');
    }
//end of code

```

Now in our routes.php we have a code like this:

```

//code starting

//routes.php
Route::get('/', function()
{
    $user = User::all();
    var_dump($user);
});
//end of code

```

Since we have four users till now, the preceding output shows them one by one throughout the key=>value pair. It starts from 0 and ends at 3.

What does it say? It says this representation is of an object of Model User class. In the User class you have defined two properties already:

```

//code snippet from User.php

/**
 * The database table used by the model
 *
 * @var string
 */

```

```
protected $table = 'users';

/**
 * The attributes excluded from the model's JSON form
 *
 * @var array
 */

protected $hidden = array('password', 'remember_token');

//code is incomplete
```

Now compare them with these two lines: `protected 'table' => string 'users' (length=5)`
`protected 'hidden' => array (size=2) ...`

I hope this comparison will inspire you to define more properties in the User class! And what can be defined is being said in the preceding output.

There are lot of properties like 'connection', 'relations', 'guarded', 'fillable', et cetera. Now hopefully you understand that these properties can be defined in your User class: not only in the User class in your Models folder, but in any class you're going to write by yourself. Suppose we have another table called 'contacts'. We can create a Contact class in our Models folder. Let us create it with few properties.

```
//code starting

//models/Contact.php

<?php

class Contact extends Eloquent{

/**
 * The database table used by the model
 *
 * @var string
 */

protected $table = 'contacts';

}
```

```
//end of code
```

Now let us go to our routes.php and change the code to this:

```
//code starting
//routes.php
Route::get('/', function()
{
    $usercontacts = Contact::all();
    var_dump($usercontacts);
});
//end of code
```

We have the same output as before like ‘users’ table, with one exception only. The only exception is in this line:

```
protected 'table' => string 'contacts' (length=8)
```

Basically, except this, every property is predefined but anytime you can override them like this. In your User class, you write something like this:

```
//code starting
//User.php
class User extends Eloquent {
protected $guarded = array('password');
}
//end of User.php
```

It means that the ‘password’ of ‘users’ table is no longer massassignable. You can even block all attributes from mass assignment like this:

```
protected $guarded = array('*');
```

So the possibilities are endless. And we will later come back to these properties and discuss in a detailed manner. Before that I would look at some typically defined query building throughout our Models. Suppose we want to find the country of the first

contacts. Laravel Eloquent ORM ships with reasonably moderate methodology that may help you to find it quite easily.

Consider this code:

```
Route::get('/', function()
{
    $usercontacts = Contact::find(1);
    var_dump($usercontacts>country);
});
```

Can we see all the contacts in a nicely formatted JSON way? Let's see:

```
Route::get('/', function()
{
    $usercontacts = Contact::all();
    foreach ($usercontacts as $key => $value) {
        echo $key . "=" . $value . "<br>";
    }
});
```

The JSON output is like this:

```
0={"id":"1","user_id":"1","address":"Address of Admin","country":"Mars","phone":"12445"}
```

```
1={"id":"2","user_id":"2","address":"Address of Sanjib","country":"Jupiter",
,"phone":"456"}
```

```
2={"id":"3","user_id":"3","address":"Address of Debangshu","country":"Moon",
,"phone":"567"}
```

```
3={"id":"4","user_id":"8","address":"Address of Mana","country":"Sun","phone":"234"}
```

It seems that Laravel has thought about it earlier and is trying to use the advantage of JSON or JavaScript Object Notation. Essentially it is a human readable way of storing the value of an array or of an object as strings. Just pick up the first value from the first row of our Contacts table and see what it says.

```
0={"id":"1","user_id":"1","address":"Address of
Admin","country":"Mars","phone":"12445"}
```

It says: our `$contact` object produces an array which has all the values of ‘contacts’ table. The first key, which is 0 obviously, represents the first row of ‘contacts’ table.

Now you may ask, why does Laravel choose JSON? There are many reasons but one of them is definitely speed. The common usage of JSON is when a front-end part of your application wants data from the back end without page load.

The same goal can be achievable by JavaScript with an AJAX request. Speed really matters when you deal with a pretty big application, doesn’t it?

It will not out of context if we discuss a bit about JSON here. I hope you would permit me to tell you something more. For a beginner it is good to learn that PHP starts serializing arrays and objects to JSON from its version 5.2.0. You may have heard about `serialize()` and `unserialize()` methods in PHP. It is nothing but converting an object to string and reversing to a new instance to get back its value.

I said JSON is a human readable way, but practically it is not really very readable as JSON stores data without white space between them. Why? As I said, speed matters! Now back to our old discussion. The output means every contact row object has been included in an array. So we can break them to get more out of it like this:

```
Route::get('/', function()
{
    $contact = Contact::all();

    foreach ($contact as $value) {
        var_dump($value);
    }
});
```

Now the output is extremely important, and it explicitly shows how Contact class behaves and creates its object. Here we see four contact objects, with each having their properties well defined. We have extended our Contact class from Eloquent and thereby we have inherited those properties.

We are interested about the first part of the preceding output:

```
object(Contact)[141]
protected 'table' => string 'contacts' (length=8)
protected 'connection' => null
protected 'primaryKey' => string 'id' (length=2)
```

```

protected 'perPage' => int 15
public 'incrementing' => boolean true
public 'timestamps' => boolean true
protected 'attributes' =>
array (size=5)
'id' => string '1' (length=1)
'user_id' => string '1' (length=1)
'address' => string 'Address of Admin' (length=16)
'country' => string 'Mars' (length=4)
'phone' => string '12445' (length=5)
protected 'original' =>
array (size=5)
'id' => string '1' (length=1)
'user_id' => string '1' (length=1)
'address' => string 'Address of Admin' (length=16)
'country' => string 'Mars' (length=4)
'phone' => string '12445' (length=5)
protected 'relations' =>
array (size=0)
empty
.....

```

You see this output is incomplete, but it serves the purpose. There are lots of properties like 'table', 'connection', 'attributes', 'original', et cetera. Each property is an array. And ultimately Laravel produces them in JSON. Now to retrieve all data you just get the key and the value will come out like this:

```
//code starting
```

```

Route::get('/', function()
{
    $contact = Contact::all();

    foreach ($contact as $value) {
        echo "Contact ID = " . $value->id . "<br>";
        echo "User ID = " . $value->user_id .
"<br>";
        echo "Address = " . $value->address .
"<br>";
        echo "Country = " . $value->country .
"<br>";
        echo "Phone = " . $value->phone .
"<br><br><br>";
        //var_dump($value) . "<br>";
    }
});
//end of code

```

And the output is quite expected. It shows every contacts detail with user ID.

//the output

Contact ID = 1

User ID = 1

Address = Address of Admin

Country = Mars

Phone = 12445

```
Contact ID = 2
```

```
User ID = 2
```

```
Address = Address of Sanjib
```

```
Country = Jupiter
```

```
Phone = 456
```

```
Contact ID = 3
```

```
User ID = 3
```

```
Address = Address of Debangshu
```

```
Country = Moon
```

```
Phone = 567
```

```
Contact ID = 4
```

```
User ID = 8
```

```
Address = Address of Mana
```

```
Country = Sun
```

```
Phone = 234
```

```
//end of output
```

The crux of Eloquent model is this. Now let us do some querying with our Eloquent Model. First we will check on the basis of user age. We have User class defined earlier. First, we want to check two users who have corresponding ages of more than 18.

```
//code starting
```

```
Route::get('/', function()
```

```
{
```

```
    $users = User::where('age', '>', 18)
    >take(2)
```

```
>get();
```

```
    foreach ($users as $user)
```



```

    {
        var_dump($user>username);
    }
});

//end of code

```

Remember, we have only three users in our users table who have ages above 18. We get the first two.

```

//output

string 'admin' (length=5)

string 'sanjib' (length=6)

//end of output

```

It produces on the basis of user ID. It goes on 1, 2, 3, and so on. If there are millions of users and you don't need to process them all to avoid eating up your memory, one good method is `chunk()`. With only four users it is difficult to show you how it works but still we can have a try and have some fun indeed.

```

//code starting

User::chunk(100, function($users)
{
    foreach ($users as $user)
    {
        //grab the $user here
    }
});

//end of code

```

Inserting, updating, or deleting with the help of Eloquent is quite easy as you have assumed already. You need not write a lot of code; instead you only write two or three lines of code to get your job done in a secured way.

```
//code starting
Route::get('/', function()
{
    $user = new User();
    $user->username = 'NewUser';
    $user->password = password_hash('pass', PASSWORD_BCRYPT);
    $user->token = 'kljhnmbgfvcdsazxwetpouytresdfhjkmnbvcdsxza';
    $user->age = 27;
    $user->created_at = time();
    $user->updated_at = time();
    $user->save();
    return 'Saved NewUser';
});
//end of coding
```

Let us see how the new user 'NewUser' has been created in our 'users' table. To see his details is very easy now. We know his age, that is, 27. So write down this code:

```
//code starting
Route::get('/', function()
{
    $users = User::where('age', '=', 27)>get();
    foreach ($users as $user) {
        var_dump($user);
    }
});
//end of coding
```

Now we have come to know how this new user object will look when we use `var_dump()` method. There are several `create()` method of Eloquent Models. You can create many users at one go. Besides, you can retrieve and update any user with a minimum amount of coding like this:

```
//code starting

Route::get('/', function()
{
    $user = User::find(2);

    $user->username = 'babu';

    $user->save();
    return 'Saved!';
});

//end of coding
```

We know that second user was 'sanjib' but now it has successfully been saved to 'babu'. Deleting is much easier than before. If we want to delete user 'NewUser' we need to write sql like this:

```
"DELETE FROM `sanjib`.`users` WHERE `users`.`id` = 10"?
```

Is it not fairly long? Because before this line you need to have to manually connect to your database, et cetera. In Eloquent Models it is much easier, as you have all things set up beforehand. The code is simple:

```
//code starting

$user = User::find(10);

$user->delete();

//end of coding
```

Before concluding our Eloquent discussion, I would like to say something about those properties again. Whenever you are going to create a new class in your Eloquent Models, you have some properties already defined and set. Laravel has done that by default. We have seen those properties before. One of them is Timestamps. By default Eloquent maintains the `created_at` and `updated_at` columns of your database automatically and you need not bother about it.

Suppose you run code like this to get all users belonging to the age less than 28 group.

```
//code starting
Route::get('/', function()
{
    $users = User::where('age', '<', 28)
>take(2)->get();

    foreach ($users as $user)
    {
        var_dump($user);
    }
});
//end of code
```

We know the output and so we don't repeat it anymore. But look at those properties, like 'timestamps', 'fillable', or 'guarded'. As long as Eloquent maintains them, it is no problem but if we want to change them according to our criteria: then what happens?

Let us see to it. Suppose we want to make timestamps 'false' and also bring some changes to 'fillable' and 'guarded' properties. The code is like this in our User class before changing the properties:

```
//code starting
//models/User.php
<?php
use Illuminate\Auth\UserTrait;
use Illuminate\Auth\UserInterface;
use Illuminate\Auth\Reminders\RemindableTrait;
use Illuminate\Auth\Reminders\RemindableInterface;
class User extends Eloquent implements
UserInterface, RemindableInterface {
use UserTrait, RemindableTrait;
```

```

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

    /**
 * The attributes excluded from the model's
JSON form.
 *
 * @var array
 */
protected $hidden = array('password',    'remember_token');
        public $timestamps = false;
        protected $fillable = array('username', 'token');
        protected $guarded = array('id', 'password');
    }

    //end of code

```

As you see, I have commented out those properties. Now we are going to change the User class, and the final code is like this:

```

//code starting

//models/User.php

<?php

use Illuminate\Auth\UserTrait;

use Illuminate\Auth\UserInterface;

```

```

use Illuminate\Auth\Reminders\RemindableTrait;
use Illuminate\Auth\Reminders\RemindableInterface;
class User extends Eloquent implements
UserInterface, RemindableInterface {
use UserTrait, RemindableTrait;

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

/**
 * The attributes excluded from the model's
JSON form.
 *
 * @var array
 */
protected $hidden = array('password', 'remember_token');
        public $timestamps = false;
        protected $fillable = array('username', 'token');
        protected $guarded = array('id', 'password');
}

//end of code

```

Database tables have relations among them. It is quite obviously true that one user would place her order knowing that the order will be added to her account. Three tables are associated instantly. Eloquent makes this managing part quite easy. In our previous database ‘sanjib’ we have two related tables: ‘users’ and ‘contacts.’ In our User class let us write a method that corresponds to the Contact Model. That is, one User has one Contact. Laravel has the method already: `hasOne()`, which passes one parameter that has relation with that Model.

```
//code starting

//models/User.php

<?php

use Illuminate\Auth\UserTrait;

use Illuminate\Auth\UserInterface;

use Illuminate\Auth\Reminders\RemindableTrait;

use Illuminate\Auth\Reminders\RemindableInterface;

class User extends Eloquent implements
UserInterface, RemindableInterface {
use UserTrait, RemindableTrait;

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

    /**
 * The attributes excluded from the model's
JSON form
 *

```

```

* @var array
*/

protected $hidden = array('password', 'remember_token');

    public $timestamps = false;

    protected $fillable = array('username', 'token');

    protected $guarded = array('id', 'password');

    public function contact()

    {

        return $this->hasOne('Contact');

    }

}

//end of code

```

In our 'users' table, user name 'mana' has user ID 8. In 'contacts' she has primary ID 4 but has a corresponding user_id 8. So our intention is to reach 'contacts' table through User Model. For that reason we have added this methodology to the User Model:

```

public function contact()

    {

        return $this->hasOne('Contact');

    }

```

Now when we write in our routes.php, code like this:

```

//code starting

//routes.php

Route::get('/', function()

{

    $contact = User::find(8)->contact;

```



```

        var_dump($contact);
    });
//end of code

```

We actually reach ‘contacts’ table through our User Model and find whose user_id is 8. The output comes out directly from the ‘contacts’ table and produces the detailed contacts of user ‘mana’

There is nothing new in it. Basically the SQL performed in this code is corresponding to this:

```

//SQL statement

select * from users where id = 8

select * from phones where user_id = 8

//end of SQL statement

```

This relationship is called One to One. You can define the inverse of the relationship on the Contact Model using belongsTo() method passing the name of the table as its first parameter. In that case in our Contact Class we will write code like this:

```

//code starting

//models/Contact.php

<?php

class Contact extends Eloquent{

/**
 * The database table used by the model.
 *
 * @var string
 */

protected $table = 'contacts';

    public function user()
    {

```

```

        return $this>belongsTo('User');
    }
}
//end of code

```

Now if we write this code in our routes.php:

```

//code starting
//routes.php
Route::get('/', function()
{
    $contact = Contact::find(4)->user;
    var_dump($contact);
});
//end of code

```

It means in our 'contacts' table we search throughout the ID and see whether it has a corresponding user_id with that. Presently, in our 'contacts' table we have corresponding user_id 10 with the contacts ID 4. This relationship means the contacts ID 4 has all the user's detail in the 'users' table and it looks upon on its corresponding user_id to see what the number is there. In this case, contacts ID 4 has a corresponding user_id 10. So it is convinced that in 'users' table it has an ID 10. In users table the ID 10 belongs to our newly created User 'NewUser'. A few pages back, we created 'NewUser' through insert method using Eloquent.

Ultimately, it brings out the output of 'NewUser'. Now you need to understand that little tricky difference between these two methods:

```

//User.php

public function contact()
{
    return $this->hasOne('Contact');
}

//Contact.php

```

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

This `hasOne()` in User Class searches for corresponding `user_id` in 'contacts' table and `belongsTo()` in Contact.php just relates its primary ID to the corresponding `user_id` in that table. Both ways, `user_id` plays the vital role and defines that One to One relationship, but, in a different way.

The same way very easily we can establish One to Many relationship among tables. Suppose one user has many posts in our 'posts' table. How can we get all of his articles in one go?

There is one method, `hasMany()`, that passes similarly parameters that can do that SQL query very easily. In our User Model first we should define that method in this way:

```
//code starting

//models/User.php

<?php

use Illuminate\Auth\UserTrait;

use Illuminate\Auth\UserInterface;

use Illuminate\Auth\Reminders\RemindableTrait;

use Illuminate\Auth\Reminders\RemindableInterface;

class User extends Eloquent implements
UserInterface, RemindableInterface {

use UserTrait, RemindableTrait;

/**

* The database table used by the model

*

* @var string

*/
```

```

protected $table = 'users';

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

    public $timestamps = false;

    protected $fillable = array('username', 'token');
    protected $guarded = array('id', 'password');

    public function contact()
    {
        return $this->hasOne('Contact');
    }

    public function post()
    {
        return $this->hasMany('Post');
    }
}

//end of code

```

And after that, we will catch every article written by any `user_id` mentioned in 'posts' table in our `routes.php` file. And the code is as follows:

```

//code starting
//routes.php

```

```

Route::get('/', function()
{
    $posts = User::find(10)>post;

    foreach ($posts as $value) {

        echo "This Post made by
User ID : {$value>user_id} <br>";

        echo "Intro: " . $value->intro . "<br>";

        echo "Article: " . $value->article
"<br>";

        echo "Tags: " . $value->tag

"<br><br><br>";

    }

});

//end of code

```

Finally, we get all the posts submitted by the 'NewUser' whose user ID is 10. Basically the articles shown here are for testing purposes. Now you can, hopefully, write your own CMS based on Laravel. Finally we will check Many to Many relationship. There are some situations where users may have different roles at the same time. Let us first add a table to our database : roles. Then assign some column like id, name, created_at, and updated_at. Now we have four roles: Administrator, Moderator, Contributor, and Member. We have had five users. Some of them have common roles: for example, user 'admin' is Administrator, Moderator, and Contributor. The user 'NewUser' is also like that. The rest of them have different roles: for example, user 'babu' is Moderator and as well as Contributor, et cetera. This is a very complex relationship but Laravel ships with a very simple solution. It has a method called belongsToMany() method that would define that relationship, but we need an intermediary table to join two tables: users and roles. That intermediary table is called 'role_user'. The name is derived from the alphabetical order of the related Model names. The table 'role_user' should have two columns: role_id and user_id. That way, you can build Many to Many relations quite easily.

```

/* We have role_user table filled up with different responsibilities. You
can run that SQL at one go to test your application. Just create the table
'role_user' in your database 'sanjib' and run this SQL: */

```

```

//role_user.SQL

```

```

-- phpMyAdmin SQL Dump
-- version 4.0.10deb1
-- http://www.phpmyadmin.net
--
-- Host: localhost
-- Generation Time: Mar 30, 2015 at 08:43 PM
-- Server version: 5.5.41-0ubuntu0.14.04.1
-- PHP Version: 5.5.21-1+deb.sury.org~precise+2

SET SQL_MODE = "NO_AUTO_VALUE_ON_ZERO";
SET time_zone = "+00:00";

/*!40101 SET
@OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;

/*!40101 SET
@OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;

/*!40101 SET
@OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;

/*!40101 SET NAMES utf8 */;

--
-- Database: `sanjib`
--
-----
--
-- Table structure for table `role_user`
--

```

```

CREATE TABLE IF NOT EXISTS `role_user` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `user_id` int(11) NOT NULL,
  `role_id` int(11) NOT NULL,
  `created_at` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON
UPDATE CURRENT_TIMESTAMP,
  `updated_at` timestamp NOT NULL DEFAULT '0000-00-00 00:00:00',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 AUTO_INCREMENT=13 ;

--Laravel Learner - 608

-- Dumping data for table `role_user`

--

INSERT INTO `role_user` (`id`, `user_id`, `role_id`, `created_at`, `updated_at`)

VALUES

(1, 1, 1, '2015-03-30 15:10:21', '0000-00-00 00:00:00'),
(2, 1, 2, '2015-03-30 15:10:46', '0000-00-00 00:00:00'),
(3, 1, 3, '2015-03-30 15:10:55', '0000-00-00 00:00:00'),
(4, 10, 1, '2015-03-30 15:11:06', '0000-00-00 00:00:00'),
(5, 10, 2, '2015-03-30 15:11:22', '0000-00-00 00:00:00'),
(6, 10, 3, '2015-03-30 15:11:31', '0000-00-00 00:00:00'),
(7, 2, 2, '2015-03-30 15:11:43', '0000-00-00 00:00:00'),
(8, 2, 3, '2015-03-30 15:11:51', '0000-00-00 00:00:00'),
(9, 8, 3, '2015-03-30 15:12:38', '0000-00-00 00:00:00'),
(10, 8, 4, '2015-03-30 15:12:45', '0000-00-00 00:00:00'),

```

```

(11, 3, 3, '2015-03-30 15:12:53', '0000-00-00 00:00:00'),
(12, 3, 4, '2015-03-30 15:12:59', '0000-00-00 00:00:00');
/*!40101 SET
CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
/*!40101 SET
CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
/*!40101 SET
COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;
//end of role_user.SQL

```

Now first of all we would set the method in our User Model like this:

```

//code starting
//models/User.php
<?php

protected $table = 'users';

protected $hidden = array('password', 'remember_token');

    public $timestamps = false;

    protected $fillable = array('username', 'token');
    protected $guarded = array('id', 'password');

    public function contact()
    {
        return $this->hasOne('Contact');
    }

    public function post()
    {
        return $this->hasMany('Post');
    }

```



```

    }

    public function role()
    {
        return $this->belongsToMany('Role');
    }
}

//end of code

```

In this method:

```

public function role()
{
    return $this->belongsToMany('Role', 'role_user');
}

```

You can explicitly pass the pivot table as the second parameter. Suppose you want to use table name 'users_role' instead 'role_user'; then you can write the following:

```

public function role()
{
    return $this->belongsToMany('Role', 'users_role');
}

```

That's it. And equally now you can use the inversion in the Role Model. The code is like this:

```

//code starting

//models/Role.php

<?php

class Role extends Eloquent{

/**

```

* The database table used by the model.

*

* @var string

*/

```
protected $table = 'roles';
```

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

//end of code

Now the time has come to test our code. Many to Many relationship works like a breeze now. The first user 'admin' has three different roles: Administrator, Moderator, and Contributor. Let us check whether in our routes we can get it or not:

```
//code starting
//routes.php
Route::get('/', function()
{
    $roles = User::find(1)->role;

    foreach ($roles as $value) {
        var_dump($value);
    }
});
//end of code
```

We are trying to find out what kind of roles the user 'admin' has. The user 'admin' has three roles: Administrator, Moderator, and Contributor. It has been shown up throughout the User Model. Now we can have an inversion of this Model. Now we are going to find out which users are Administrator or Moderator or Contributor or Member. Like, in our routes.php we can write the code this way:

```
//code starting

//routes.php

Route::get('/', function()
{
    $roles = Role::find(1)->user;

    foreach ($roles as $value) {
        var_dump($value);
    }
});

//end of code
```

In this code, we actually want to find out who the Administrators are, because role ID 1 is Administrator. Here are two users who are Administrators. They are 'admin' and 'NewUser'. The detailed data about them has been taken out from the 'users' table. From 'roles' table through the Role Model, you can search any role that has been assigned to the users. There could be thousands of Members or Contributors. But in a few lines of codes, Laravel makes your task much simpler. In the next chapter, we will see some more features of Laravel that deal with database.



How Security and Authentication Work Together

Laravel implements authentication in such a simple way that you have everything just out of the box. To do that, Laravel returns an array in 'app/config/auth.php' where these things are defined and it also well documented. In this file it is clear that the driver will be Eloquent, the model will be User, and the table is 'users'. There is another table: 'password_reminder'. We will discuss it later. So by default it has been defined; now all you have to do is use the static methods Laravel ships with. To make your password stronger, Laravel has 'Hash' class that provides secure Bcrypt hashing. You can make your password stronger with this method:

```
$password = Hash::make('yourpassword');
```

We can check how it works. Let us go to our routes.php and first find what the password of the user 'NewUser' is.

```
//code starting  
  
//routes.php  
  
Route::get('/', function()  
{  
    $user = User::find(10);  
    echo $user->password;  
  
});
```

```
//end of code
```

The output is:

```
$2y$10$XTne1FY0oTLqGYPIiu/7F.4AxUae9akpjpFV.xizdq3IytI9N1Nim
```

We make this password with PHP `password_hash()` method like this:

```
$user->password = password_hash('pass', PASSWORD_BCRYPT);
//end of password_has() method
```

Now we can update this password with Laravel's own `Hash::make()` method and see how it looks like. First update the password:

```
//code starting
Route::get('/', function()
{
    $user = User::find(10);

    $user->password = Hash::make('pass');

    $user->save();

});
//end of code
```

Now the output looks like:

```
//output
$2y$10$NI9gu2x3q78dzE2J.h9kweJaD4I1M27kHokiZ.yfABynpABbNA/zW
//end of output
```

We can check the length of the password. It is a string and length is 60. If we `var_dump($user)` in our `routes.php` we get all detail about the user 'NewUser' through the User model, and the output will state the password length.

We can again go back to our old `password_hash()` method and change the password of user 'NewUser' like this:

```
//code starting
//routes.php
```

```
Route::get('/', function()
{
    $user = User::find(10);

    $user->password = password_hash('pass', PASSWORD_BCRYPT);

    $user->save();
});
//end of code
```

Run the code and again we `var_dump($user)` and see the output of password now:

```
//output
'password' => string
'$2y$10$zjBulZIfVmMZu/QDGdoCROoKlQysWs/hAVKFTGLNp60EiG5K5z1
W0' (length=60)
//end of output
```

The password has been updated. The length is 60 anyway. You may wonder why I spend so much time on the discussion of passwords! First of all, as far as security is concerned, you need to be careful about passwords. You should make them stronger. What I wanted to prove here is that Laravel always uses the most advanced technology available in PHP. The same result is available through code with the help of Laravel. Now we will try some authentication process using the default interfaces and methods Laravel ships with.

As of we find some methods that point to the authentication process. We can use them in our User Model like this:

```
//code starting

//models/User.php

<?php

use Illuminate\Auth\UserTrait;

use Illuminate\Auth\UserInterface;

use Illuminate\Auth\Reminders\RemindableTrait;
```

```

use Illuminate\Auth\Reminders\RemindableInterface;

class User extends Eloquent implements
    UserInterface, RemindableInterface {
    use UserTrait, RemindableTrait;

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

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

    public $timestamps = false;

    protected $fillable = array('username', 'token');

    protected $guarded = array('id', 'password');

    public function getAuthIdentifier(){
        return $this->getKey();
    }

    public function getAuthPassword() {
        return $this->password;
    }
}

```

```

public function post()
{
    return $this->hasMany('Post');
}

public function contact()
{
    return $this->hasOne('Contact');
}

public function role()
{
    return $this->belongsToMany('Role', 'role_user');
}
}

//end of code

```

Now you can create a ‘dashboard’-type authentication system for our administration. It is fairly simple and short but you get the idea and I hope you can build your ‘CRUD’ based on this concepts.

Now it is time to use ‘Auth’ and ‘Session’ class and necessary static methods in our ‘routes.php’ to make this authentication mechanism successful.

```

//code starting

//routes.php

::get('/', 'HomeController@home');

Route::get('/login', 'HomeController@login');

Route::post('login', function(){

    if(Auth::attempt(Input::only('username', 'password'))){

        return Redirect::intended('/');
    }
}

```



```

    } else {

        return Redirect::back()->withInput()->with('error', "Invalid
        credentials");

    }

});

Route::get('logout', function(){

    Auth::logout();

    return Redirect::to('/')->with('message', 'You are now logged out');

});

Route::group(array('before'=>'auth'), function(){

    Route::get('/', 'HomeController@home');

    Route::get('user/{id}', 'HomeController@user');

});

//end of code

```

Now let us explain the steps one by one: First we need to have a protected 'index' page. To do that all we need to do is use the following:

```

Route::group(array('before'=>'auth'), function(){

    Route::get('/', 'HomeController@home');

    Route::get('user/{id}', 'HomeController@user');

});

```

By this group routing we can define which pages are only for authenticated users. We have defined two pages as per our planning: 'index' and 'users'.

We can also use the `Auth::logout()` method to end the session, it redirects the user to the 'index' page again, which is nothing but the 'login' page. We will conclude this chapter with a few authentication methods that Laravel ships with by default. For passwords, you can use these methods:

```

//code starting

Hash::make('yourpassword');

```

```
Hash::check('yourpassword', $hashedPassword);  
Hash::needsRehash($hashedPassword);  
//end of code
```

And for authentication, you can use these methods:

```
//code starting  
Auth::check();  
Auth::user();  
Auth::attempt(array('email' => $email, 'password'=> $password));  
Auth::attempt($usercredentials, true);  
Auth::once($usercredentials);  
Auth::login(User::find(1));  
Auth::loginUsingId(1);  
Auth::logout();  
Auth::validate($usercredentials);  
Auth::basic('username');  
Auth::onceBasic();  
Password::remind($usercredentials,  
function($message, $user){});  
//end of code
```

CHAPTER 18



How Request, Response Work in Laravel 5

What Does Request Mean?

When a user requests some pages, what happens? You have just learned that in the previous chapter. We can obtain that request instance and see how it looks. To do that we would like to make a kind of typehinting our 'Illuminate\Http\Request' class on our controller constructor or method. Suppose we have a 'TestController'.

So the code will look like this:

```
//code starting

<?php namespace App\Http\Controllers;

    use Illuminate\Http\Request;

    use Illuminate\Routing\Controller;

class TestController extends Controller
{
    protected $request;

    /**
     * @param Request $request
     * @return Response
     */
    public function testRequest(Request $request)
    {
```

```

        $this->request = $request;

        $uri = $request->path();

        var_dump($uri);
    }
}

```

//code ended

Besides we have a route in our 'routes.php' like this:

//code starting

```
get('test', 'TestController@testRequest');
```

//code ended

Now if we go to our browser and type <http://localhost:8000/test> we get a response like this:

```
string 'test' (length=4)
```

If we type in our browser like this:

<http://localhost:8000/test/hagudu/class=5/school=Don%20Bosco> and in our route level we pass two more variables like this:

//code starting

```
get('test/{name}/{class}/{school}', 'TestController@testRequest');
```

//code ended

We get an output like this:

```
string 'test/hagudu/class=5/school=Don%20Bosco'
(length=38)
```

So your user sends a request and we trace the URL path through our request object. In Laravel 5 this is done by dependencies injection. We will discuss this in great detail. So if you are an absolute beginner, just don't get frightened. It appears we can do lot of stuff through this Request object. We can send some inputs through a form and receive it in our Controller and update User's profile. We can manage payments and many more.

Now if want the full URL path we can type in our controller like this:

```
$url = $request->url();
```

And the output changes to this:

```
string 'http://localhost:8000/test/hagudu/class=5/school=Don%20Bosco'
(length=60)
```

You get the full URL instead of only URI path. There are many Request methods you will find very useful later. Since you have not started Laravel yet, we don't want to dig very deep. There are few methods you may just want to know, such as this:

```
$name = $request->input('name');
```

In this case, you get a name from your form fields. You can get an array also, and the retrieval is not very complex.

```
$input = $request->only('username', 'password');
$input = $request->except('credit_card');
```

In this case you only get the user name and password except the credit card information. There are a lot more examples, and we will come to them when proper time comes.

What Does Response Mean?

Basically, what all routes and controllers send back to the browser is Response. Users send Requests and the Server sends back Response. The basic mechanism is simple enough to understand what is going on under the hood. Laravel can do many things and is equipped with several ways to send Responses.

But the foremost Response is string in a Route like this:

```
Route::get('/', function () {
    return 'Hello World';
});
```

But this is a pretty simple string. Laravel can do much more, like returning an 'Illuminate\Http\Response' instance or a complete View. What does this returning a full 'Response' instance mean? It actually allows you to customize the response's HTTP status code and headers. A 'Response' instance inherits from the 'Symfony\Component\HttpFoundation\Response' class, providing a variety of methods for building HTTP responses. After installing Laravel, if you open the 'vendor/symfony' directory, you will not find the 'Symfony\Component\HttpFoundation\Response' class that Laravel inherits there. But if you ever individually work with and install 'Symfony' components you will find that class.

The HttpFoundation defines an objectoriented layer for the HTTP specification. It provides an abstraction for requests, responses, uploaded files, cookies, sessions, et cetera. You won't get it inside Laravel, but for your knowledge and understanding just know that we use Symfony components like this:

```
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
$request = Request::createFromGlobals();
echo $request->getPathInfo();
```

And we use the Response object like this:

```
$response = new Response('Not Found', 404, array('ContentType' => 'text/
plain'));
$response->send();
```

And in Laravel we basically return Response like this:

```
use Illuminate\Http\Response;

Route::get('home', function () {
    return (new Response($content, $status))->header('ContentType',
    $value);
});
```

You may send a JSON Response like this:

```
Route::get('/', function () {

    return response()->json(['name' =>
'Sanjib', 'location' => 'Pluto']);

});
```

And you have an output like this on your browser:

```
{"name":"Sanjib","location":"Pluto"}
```

A pure JSON output is a readable human format. Another good example of Response is Redirect. You can use two helper methods to Redirect users as necessary. When you use Redirect responses, they are actually instances of the 'Illuminate\Http\RedirectResponse' class. They contain the proper headers needed to redirect the user to another URL. There are several ways to generate a 'RedirectResponse' instance. The simplest method is to use the global 'redirect' helper method:

```
Route::get('yourpage', function () {
```

```
return redirect('/yourpage');  
});
```

And there are also back methods that also come ready to hand. If you want to make your user fill up certain forms and want to validate that with input, you can use something like this:

```
Route::post('your/profile', function () {  
    // Validate the request  
    return back()->withInput();  
});
```

When you learn more about named routing and controller actions, you may want to use them like this:

```
return redirect()->route('login');
```

Or like this:

```
return redirect()->action('HomeController@index');
```

CHAPTER 19



Contracts vs. Facades

By so far you have understood that the total Laravel 5 framework depends on many blocks of Interfaces, classes, and other packaged versions that package developers have developed so far. Laravel 5 has happily used them, and I encourage you to do the same by following the SOLID principle and loose coupling. To master the Framework properly, you need to understand the core services that run Laravel 5. What are the main resources behind the scene? Basically, Contracts come in between regarding this scenario.

Contracts are interfaces that provide this core services. Like 'Illuminate\Contracts\Mail\Mailer' defines the process of sending mails and doing that, this interface simply pools in the implementation of mailer classes powered by the SwiftMailer. Now what are the 'Facades'? This chapter's heading is 'Contracts vs. Facades'. Do they have any similarity or relationship or anything else? Let me mention something about Facades first.

Facades are also interfaces. But they have a distinct difference from Contracts. First of all, Facades are static interfaces that supply methods to the classes of service container. You have already seen a lot of Facades already. Remember 'App', 'Route', 'DB', 'View', et cetera. The main quality of Facades is that you can use them without typehinting. Like in your 'routes.php' file you can write this code:

```
Route::bind('books', function ($id){  
  
    return App\Book::where('id', $id)->first();  
  
});
```

This 'Route' Facade directly resolves the contracts out of service container. Though Facades are static interfaces, they have more expressive syntaxes and provide more testability and flexibility than 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 innovate something more, you need to use Contracts. How you could 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 'BookController' through which you want to maintain a long list of your favorite books. To do that you need to store books in database. To do that you can bind your 'Book' Model in your 'routes.php' first, and then using resource you can do all kinds of CRUD operations. In doing so, you need to log in.

Consider this code:

```
public function store(Request $request)
{
    if (Auth::check()) {
        // The user is logged in
    }
}
```

You can check whether the user is logged in or not. And depending on that you can add only your favorite books. For checking you use ‘Auth’ Facade. That’s fine; As long as you don’t want more decoupled state it works great. But if you’d rather follow the SOLID principle and want a more decoupled state then instead of depending on concretion you should adopt a more robust abstract approach. And in that case Contract comes to your rescue. Taylor Otwell himself keeps github repositories on Contracts, so you better have a look around it. All of the Laravel Contracts live there:

<https://github.com/illuminate/contracts>

Now let us go back to our previous topic. So we can either use our ‘Auth’ Facade. But try to understand one thing; when you use facade it assumes the framework and is very tightly coupled with a concretion. But considering the SOLID principle, you want a decoupled state. What to do? Let us consider another scenario. We can inject our ‘Auth’ Facade through constructor like this and rewrite our whole code this way:

```
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 you might opine that it is much better as we have injected ‘Auth’ instance through our constructor. Yes, it is better than before but still it lacks the SOLID design principle. It depends upon a concrete class like the following:

```

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 see, when we use this line of code in our ‘BookController’: `use Illuminate\Auth\Guard as Auth`, we actually inject an instance based on concretion not abstraction. When we unit test our code, we will have to rewrite our codes. Moreover, whenever you call any methods through this instance it is aware of your framework. But we need to make it completely unaware of our Framework and become loosely coupled. To do that we just have to change one line of code in our ‘BookController’.

Instead of that line of code:

```
use Illuminate\Auth\Guard as Auth;
```

We write this line of code:

```
use Illuminate\Contracts\Auth\Guard as Auth;
```

And that is it! Now our ‘Auth’ instance is completely loosely coupled. And now we can change that line of code in `store()` method:

```
if (Auth::check()) {
    // The user is logged in
}
```

to this line of code:

```
$this->auth->attempt();
```

And it is done. Now your application has achieved more sophistication by following the SOLID design principle and becomes completely loosely coupled. Finally, if we were to check the interface `Illuminate\Contracts\Auth\Guard`, what would we see? Just have a look, so that you can understand what happens behind the scenes. The code of that interface is pretty big, so for brevity we just cut the `attempt()` method out of it. The interface looks like this:

```
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
```

Now your code is not coupled to any vendor or even to Laravel. You are not compelled to follow a strict methodology by a concrete class. You can simply implement your own, alternative methodology out of any contract.

I hope this comparison between “contract” and “facade” makes sense.

CHAPTER 20



Middleware, Layer Filter, and Extra Security

HTTP Middleware is one of the best facilities Laravel 5 ships with. It not only adds extra security to your application but also gives you enough freedom to create your own security mechanism alongside the default Laravel Authentication mechanism. As you know already, when a user requests for a page, the browser sends the request and the server responds. Sometimes, this request-response mechanism is simple and sometimes it is fairly complicated. But at the end of the day whenever a user requests for a page a HTTP request enters your application. Most of the time it is innocuous, but as the proverb goes you cannot and should not rely on user's input or request so it needs to be filtered. It has to be filtered when your application needs an extra bit of authentication or security measures to be taken. Middleware does this out of the box.

Laravel 5 ships with 'Authenticate' middleware. It is default so that you need not tweak it to add some extra security to your application. The mechanism is very simple. It verifies the user's credentials and permits it to enter your application and proceed further. Besides this default Middleware we can add our own functionalities.

Next, in the construct level it instantiates this 'auth' so that in the handle function it can check whether the user is a guest or an authenticated user. A fairly simple task, but there is something more. In that folder, there is another file: 'RedirectIfAuthenticated.php'. Let us see its code also:

```
<?php

    namespace App\Http\Middleware;

use Closure;

use Illuminate\Contracts\Auth\Guard;

class RedirectIfAuthenticated
{

    /**
```

```

    * The Guard implementation.
    *
    * @var Guard
    */
protected $auth;
/**
    * Create a new filter instance.
    *
    * @param Guard $auth
    * @return void
    */
public function __construct(Guard $auth)
{
    $this->auth = $auth;
}
/**
    * Handle an incoming request.
    *
    * @param \Illuminate\Http\Request $request
    * @param \Closure $next
    * @return mixed
    */
public function handle($request, Closure $next)
{

```

```

    if ($this->auth->check()) {
        return redirect('/home');
    }

    return $next($request);
}
}

```

It does almost the same thing. It is checking user's authentication and permits it to go to the desired location. Here it is:

```

1. if ($this->auth->check()) {
2.     return redirect('/home');
3. }

```

Now, the question is, how does it work out of the box? To find out our answer, we need to go to our 'routes.php' page. Here we have already seen how we can use the 'log in, log out, and register' process using the default authentication mechanism. Doing that, we had to write a piece of code like this:

```

1. Route::group(['middleware' => 'auth'], function () {
2.     Route::get('home', function () {
3.         return view('auth.home');
4.     });
5.
6.     Route::get('dashboard', function () {
7.         return view('auth.dashboard');
8.     });
9. });

```

You go through the first line and hopefully it now makes sense with the default Middleware logic. In our Route Group, we used

```
['middleware' => 'auth']
```

And it distantly connects with the in-built Authentication Middleware. Actually it is defined in 'app/Kernel.php' as a protected property.

```

1. /**
2.     * The application's route middleware.
3.     *
4.     * @var array
5.     */
6.     protected $routeMiddleware = [
7.         'auth' =>
\App\Http\Middleware\Authenticate::class,
8.         'auth.basic' =>
\Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
9.         'guest' =>
\App\Http\Middleware\RedirectIfAuthenticated::class,
10.    ];

```

How the default Authenticate Middleware works is now clearcompiled. How about adding some our own functionalities? We saw in our Authentication chapter that a user logs in and she goes to the '/dashboard'. Well, we need to add some extra functionalities in our 'users' table first, so that one of the users will be the administrator and only she can check into a 'blog' page.

Since we need to test our own Middleware at the ground Laravel, we have decided to make it the simplest one. So we keep a link to that 'Blog' page in our 'Dashboard' page. Now every user logs in and clicks that link. But our Middleware will check whether that user is administrator or not. If she is an administrator, she can land on the 'Blog' page with special Administrative power, and if she is not, she will sent back to the home page for general users. This is our simple administrator Middleware building procedural step. Let us start.

First, we need to set up one extra column in our 'users' tablecell. We need to migrate for that. You can do it manually in your PHPMyAdmin or directly use Tinker to add that column. But migration is simpler.

Open up the terminal, go to our Laravel application, and issue this command:

```
php artisan make:migration add_is_admin_to_user_table --table=users
```


Go to the 'database' folder and see that your new migration has been created. Now before migrating starts we need to tweak the up() and down() method like this:

```

1. <?php
2.
3. use Illuminate\Database\Schema\Blueprint;
4. use Illuminate\Database\Migrations\Migration;
5.
6. class AddIsAdminToUserTable extends Migration
7. {
8. /**
9.     * Run the migrations.
10.    *
11.    * @return void
12.    */
13.    public function up()
14.    {
15.        Schema::table('users', function (Blueprint $table) {
16.            $table->boolean('is_admin')->default(false);
17.        });
18.    }
19.
20.    /**
21.     * Reverse the migrations.
22.     *

```

```

23.     * @return void
24.     */
25.     public function down()
26.     {
27.         Schema::table('users', function (Blueprint
                $table) {
28.             $table->dropColumn('is_admin');
29.         });
30.     }
31.
}
```

Now we are ready to migrate, so issue this command:

```
php artisan migrate
```

Check your ‘users’ table; you’ll see that one extra column has already been added, and each one is false. So you choose any one user and make her the administrator so that when we check the authentication it comes out true. Our database setup is ready. Now we can proceed to the next step. This step involves creating our own Middleware logic which will check whether the user is administrator or not. If she is the administrator, she can go the ‘Blog’ page. Otherwise she will sent back to the Home page for general users. We can create our custom Middleware through the console. In our terminal, we issue this command:

```
php artisan make:middleware RoleMiddleware
```

As you see, we name our Middleware ‘RoleMiddleware’. You can name it differently. No problem. It will immediately shoot back with a prompt that says that your middleware has been successfully created. Let us go to the ‘app/Http/Middleware’ folder and check it. Yes, it has been created. It comes up with a handle() method as usual. Now all we need is to add this line of code on top of the page first:

```
use Illuminate\Contracts\Auth\Guard;
```

We need to check the user’s authenticity. The full code looks like this:

```
1.<?php
```

```
2.
3. namespace App\Http\Middleware;
4. use Closure;
5. use Illuminate\Contracts\Auth\Guard;
6. use Illuminate\Http\RedirectResponse;
7. //use App\User;
8.
9. class RoleMiddleware
10.
11. {
12.     protected $auth;
13.
14.     public function __construct(Guard $auth) {
15.         $this->auth = $auth;
16.     }
17.
18.     /**
19.      * Handle an incoming request.
20.      *
21.      * @param \Illuminate\Http\Request $request
22.      * @param \Closure $next
23.      * @return mixed
24.      */
```

```

25.     public function handle($request, Closure $next)
26.     {
27.         if ($this->auth->check())
28.         {
29.             if ($this->auth->user()->is_admin == TRUE)
30.             {
31.                 return $next($request);
32.             }
33.         }
34.
35.         return new
36.             RedirectResponse(url('/auth/login'));
37.     }
38. }

```

Lines 27 and 29 are very important; in fact, it ultimately plays the crucial part in our own Middleware.

The logic is fairly simple. Our 'RoleMiddleware' class is called up whenever an instance of 'Guard' is created and that will take care of further authentication process. So in our handle() method that passes two parameters, 'request' and 'Closure \$next', we will add some extra spices so that when it checks the credentials of the user, it looks up and asks if she is the administrator. If not, it sends her back to the login page. So if someone just types the URL and try to enter into our 'Blog' page she will be sent back to the login page. Now if she is a registered user she can log in but ends up at the 'Dashboard' page. To facilitate this 'RoleMiddleware' we need to add an extra Route Group in our 'routes.php' file like this:

```

1. Route::group(['middleware' => 'auth'], function () {
2.     Route::get('home', function () {
3.         return view('auth.home');
4.     });

```

```

5.
6.     Route::get('dashboard', function () {
7.         return view('auth.dashboard');
8.     });
9. });
10.
11.     Route::group(['middleware' => 'role'], function ()
12.     {
13.         Route::get('blog', function () {
14.             return view('blog.index');
15.         });
16.     });

```

You see at line 10, we add an extra Route Group Middleware, 'role'. But the question is how our application will learn about it. Okay, our steps have been incomplete till now. We need to register it to 'app/Kernel' so the 'app/Kernel.php' code looks like this:

```

1. protected $routeMiddleware = [
2.     'auth' => \App\Http\Middleware\Authenticate::class,
3.     'auth.basic' => \Illuminate\Auth\Middleware\
4.         AuthenticateWithBasicAuth:
5.         :class,
6.     'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
7.     'iplog' => \App\Http\Middleware\RequestLogger::class,
8.     'role' => \App\Http\Middleware\RoleMiddleware::class,
9. ];

```

Watch out for line 6. We have registered our

```
'role' => \App\Http\Middleware\RoleMiddleware::class,
```

And it will now work like a breeze. The next steps are left to create one Controller and View page. That you can do. In the next chapter we will officially launch a administrator facility and will create a Administrative dashboard where an administrator can create, edit, and delete users.

So far you have seen how we can control access to the users by using various types of Middleware. Use your imagination to build up all the crazy Middleware, and the procedure is simple enough. You can handle it by yourself. Suppose we want to check the points users have earned in a forum. We can do some more weird staff like checking whether a user's name begins with any particular letter, et cetera. We can use 'before' and 'after' Middleware so that our application will handle something beforehand and execute something after.

Suppose we want to check the forum points of a user. We can create a Middleware like this:

```
1. namespace App\Http\Middleware;
2.
3.     use Closure;
4.
5.     class PointMiddleware
6.     {
7.         /**
8.          * Run the request filter.
9.          *
10.         * @param
11.         \Illuminate\Http\Request $request
12.         * @param \Closure $next
13.         * @return mixed
14.         */
15.         public function handle($request, Closure $next)
```

```

15.     {
16.         if ($request->input('point') <= 500) {
17.             return redirect('home');
18.         }
19.
20.         return $next($request);
21.     }
22.
23.
}

```

Now we can also handle this before and after like this:

```

1. namespace App\Http\Middleware;
2.
3.     use Closure;
4.
5.     class BeforeMiddleware
6.     {
7.         public function handle($request, Closure
8.             $next)
9.         {
10.             // Perform action
11.
12.             return $next($request);
13.         }

```

```

14.
    namespace App\Http\Middleware;
15.
16.     use Closure;
17.
18.     class AfterMiddleware
19.     {
20.         public function handle($request, Closure
                $next)
21.         {
22.             $response = $next($request);
23.
24.             // Perform action
25.
26.             return $response;
27.         }
28.     }

```

So there are lots of actions waiting to be explored inside this Middleware section. We will conclude with a logging Middleware that will log every request coming to your application: very simple and elegant. You have learned how to create a Middleware. So create a 'IPLogger' Middleware first. And add one line of code in your handle() method like this:

1. <?php
- 2.
3. namespace App\Http\Middleware;


```

4.
5. use Closure;
6.
7. class IPLogger
8. {
9.     /**
10.      * Handle an incoming request.
11.      *
12.      * @param \Illuminate\Http\Request $request
13.      * @param \Closure $next
14.      * @return mixed
15.      */
16.     public function handle($request, Closure $next)
17.     {
18.         \Log::info($request->getClientIp());
19.         return $next($request);
20.     }
21. }

```

Next in your 'routes.php' add this Middleware Group:

```

1. Route::get('/', ['middleware' => 'iplog', function ()
2.     {
3.         return view('index');

```

So that each time a user sends request to the index page, his 'ip address' will be stored in the 'storage/logs/laravel.log' like this:

1. [20150822 18:16:14] local.INFO: ::ipaddress
2. [20150822 18:16:55] local.INFO: ::ipaddress

I should mention that you should have registered your 'IPMiddleware' in 'app/Http/Kernel.php' like this:

```
'iplog' => \App\Http\Middleware\IPLogger::class,
```

And it is done!

Please have a look into the documentation for more valuable inputs like Middleware parameters and Terminable Middleware, where you start a session, which is stored and terminated after your task has been accomplished.

Index

■ A

- Abstraction, 18
- Anonymous functions, 28
- Authentication and authorization, 81
 - AuthController, 81–84
 - config/app.php, 81
 - database/migration file, 85
 - login.blade.php, 85–86, 88–89
 - logout option, 89–90
 - validator() method, 84

■ B

- Blade. *See* Views and blade

■ C

- Composer
 - centrally/globally, 2
 - commands, 2–3
 - file type, 3
 - globally, 4
 - home page, 1
 - installation of Laravel 5.3, 4
 - Internet connection, 4
 - Laravel 5.2, 3
 - locally, 2
 - monolog package, 3
 - start page, 1–2
- Contracts *vs.* Facades
 - attempt() method, 170
 - BookController method, 170
 - CRUD operations, 167
 - interfaces, 167
 - SOLID principle, 168–169
 - store() method, 170

- Controller class, 29
 - controller, 35
 - IoC container, and interface, 35
 - layers, 29
 - MyController.php file, 30–31
 - page view, 31
 - presentation layer, 29
 - resourceful controller, 34
 - RESTful, 32
 - role of, 33
 - routes.php file, 29–30
- Cross-site request forgery (CSRF), 25
- CRUD (create, retrieve, update, delete)
 - application, 59
 - controller method, 70
 - create.blade.php, 74–76
 - creation (review page), 74
 - edit page, 77, 79
 - editreview.blade.php page, 76
 - index.blade.php page, 70–72
 - index page, 79
 - page view, 76
 - review page, 70, 72
 - review.php file, 68
 - routes.php file, 73, 76
 - show.blade.php, 73
 - table.php file, 67–68
 - TaskController.php file, 68, 69, 72, 74
 - url function, 72

■ D

- Database migration, 49
 - advantage, 49
 - .env file, 49
 - migrate command, 51
 - PHP file, 50–51

Database migration (*cont.*)

- phpMyAdmin tool, 52
- tasks table, 50
- up and down function, 50
- users table, 49

■ **E**

Eloquent relations

- advantage of, 128
- belongsTo() method, 141
- chunk() method, 133
- contact class, 126–127
- contact() method, 140
- contacts details, 131–132
- contacts table, 129
- creation, 53
- crux of, 132
- database tasks table, 57, 124
- definition, 123
- empty class, 56
- final source code, 137, 139
- function(), 129
- hasMany() method, 143–144
- hasOne() method, 139, 143
- insert, update/delete, 133–134
- interface, 54
- JSON format and
 - output, 128
- mass assignment, 56, 127
- model class, 53
- NewUser method, 134, 142–143
- ORM, 123
- output result, 129–131
- PHP file, 53
- posts table, 144–145
- properties, 127, 136
- role() method, 149–150
- role_user table, 145–149
- routes.php, 125, 140, 151
- save() method, 54–55
- serialize() and unserialize()
 - methods, 129
- SQL statement, 141
- tables creation, 123
- tasks table, 54
- types of, 123
- update() method, 55
- user class, 124–125

- User.php, 125–126
- var_dump() method, 135–136

■ **F, G**

Facades, 167

File structure

- app folder, 15
- bootstrap folder, 16
- database folder, 16
- home page, 17
- installation, 16
- interfaces and method, 19
- Laravel 5.3.18, 15
- public folder, 16
- SOLID design principle, 17

Forms, 65

- aliases array, 65
- providers array, 65

■ **H**

Homestead

- folder and site section, 11
- Homestead.yaml file, 10
- installation and configuration, 10
- test.app, 13
- Vagrant box and Vagrant, 9
- vagrant up command, 12

■ **I, J, K**

Interfaces and method injection, 19

■ **L**

Laravel Homestead, 7

■ **M**

Middleware, layer filter and security, 173

- app/Kernel.php file, 176, 181
- authentication, 173
- blog page, 176
- construct level, 173
- handle() method, 178–180, 184–185
- middleware, 182–184
- RedirectIfAuthenticated.php, 173

- RoleMiddleware, 178
- routes.php file, 175, 180–181, 185
- storage/logs/laravel.log, 186
- up() and down() method, 177–178
- Model, view, controller and workflow, 59
 - all() method, 60
 - routes.php file, 59
 - task.blade.php page, 61
 - TaskController.php file, 60
 - task model class, 60
- ModelViewController methodology, 123

■ N, O

- Named routes, 26

■ P, Q

- PHP 5, 19

■ R

- Representational State
 - Transfer (REST), 32
- Request
 - meaning, 161
 - output, 163
 - TestController, 161–162
 - URI path, 163
- Resourceful controller, 34
- Response method
 - components, 163
 - helper method, 164–165
 - JSON, 164
 - meaning, 163
 - output, 164
 - Route function(), 163
- returningASimplePage function, 30
- Routing, 21
 - anonymous functions, 21, 28
 - any() method, 25
 - app/Http/routes.php file, 21
 - best practices, 25
 - concept, 28
 - CSS style, 22
 - HTML code, 22
 - HTTP protocol, 21
 - methods, 24
 - named routes, 26
 - organize files, 27
 - routes.php file, 21, 24
 - web browser, 23

■ S, T, U

- Security and authentication, 153
 - Auth, 158–159
 - index page, 158
 - make() method, 154
 - NewUser method, 153
 - password_hash() method, 154
 - password_reminder table, 153
 - routes.php, 157–158
 - User model table, 155–157
 - var_dump method, 155
- SOLID design principle, 17
- SQLite database, 63
 - database.sqlite file, 64
 - .env file, 63
 - file path, 63
- Staticmethod. *See* Routing

■ V, W, X, Y, Z

- Vagrant, 7. *See also* Homestead
- Validation
 - action.php, 96
 - conditional rules, 112–113
 - custom validation, 113–118
 - error messages, 104–107
 - \$error variable, 109
 - form, 91–92
 - validation, 119–120, 122
 - view page, 93
 - meaning, 91
 - MyCustomValidation class, 117
 - parameters, 105
 - register.blade.php, 108–109
 - registration.blade.php file, 92
 - rules, 110–112
 - source code, 97–103
 - validate.php class, 94–95
- Views and blade
 - about.blade.php file, 43, 44
 - compact method, 45
 - CSS style code, 42
 - dynamically passed data, 46
 - master.blade.php code, 41
 - MyController code, 45
 - process, 41
- Virtual box, 7
 - download section, 8
 - installation, 7
 - Kali Linux and Windows XP, 9
 - UBUNTU software, 8