



C o m m u n i t y E x p e r i e n c e D i s t i l l e d

Learning Laravel 4 Application Development

Develop real-world web applications in Laravel 4 using its refined and expressive syntax

Hardik Dangar

[PACKT] open source*
PUBLISHING community experience distilled

Learning Laravel 4 Application Development

Develop real-world web applications in Laravel 4 using its refined and expressive syntax

Hardik Dangar



BIRMINGHAM - MUMBAI

Learning Laravel 4 Application Development

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: December 2013

Production Reference: 1171213

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

ISBN 978-1-78328-057-5

www.packtpub.com

Cover Image by Aniket Sawant (aniket_sawant_photography@hotmail.com)

Credits

Author

Hardik Dangar

Reviewers

Sean Mumford
Michele Somma
Rahul Taiwala
Samuel Vasko

Acquisition Editors

Martin Bell
Saleem Ahmed
Subho Gupta

Lead Technical Editor

Arun Nadar

Technical Editors

Tanvi Bhatt
Mrunmayee Patil

Copy Editors

Roshni Banerjee
Brandt D'Mello
Janbal Dharmaraj
Tanvi Gaitonde
Mradula Hegde
Gladson Monteiro
Deepa Nambiar
Shambhavi Pai

Project Coordinator

Shiksha Chaturvedi

Proofreader

Mario Cecere

Indexer

Mehreen Deshmukh

Graphics

Abhinash Sahu

Production Coordinator

Arvindkumar Gupta

Cover Work

Arvindkumar Gupta

About the Author

Hardik Dangar is co-founder of Sphere Rays Technolabs Pvt. Ltd. His primary focus has been on web-based ERP systems for the last six years. Lately, he has been working with a few of the key startups to shape their core ideas into architectural development and implementation of projects from inception. He can be found on Google+ developer communities and IRC channels of various frameworks helping people and building communities.

First of all, thanks to the entire team at Packt for helping me through the entire process of book writing. I would like to thank Martin Bell for helping me to get through the initial nerves when I started. I would also like to thank Saleem Ahmed for the later part of the book and Angel Jathanna for being helpful in making my schedule work and bearing with my schedules. And thanks to all the other people who have worked incredibly hard to make this possible.

I would like to thank Taylor Otwell for crafting Laravel – the most elegant framework I have ever worked on. Laravel has changed the way I do Web development.

I'd also like to thank Preetam and Anup for doing a third-person review of the book and making many helpful suggestions.

Last, but not the least, thanks to my entire family for their love, support, and patience. Many thanks to my wife Jagruti for her understanding and support during the writing of this book.

About the Reviewers

Sean Mumford is a web developer currently writing HTML, CSS, JavaScript, and object-oriented PHP in Nashville, Tennessee. He has a decade of experience writing code and eight years of experience in writing code that doesn't suck. He currently develops web applications on behalf of Tennessee's taxpayers at the TN Secretary of State's office. He occasionally dabbles in freelance work and unrealistically ambitious side projects.

I'd like to thank Taylor Otwell, Sean McCool, Jeffery Way, and the other community leaders for making Laravel the framework it is and helping me become a better developer. I'd also like to thank the author for taking the time and energy to write a book on developing with Laravel, and Packt for having the confidence in my knowledge and experience to include me as a technical reviewer.

Michele Somma is an Italian web developer skilled in PHP, MySQL, and some new frameworks such as jQuery, jQuery UI, and Twitter Bootstrap. He has been a major user of PHP CodeIgniter Framework for over two years and recently migrated to the new Framework Laravel. He likes to develop both frontend and backend application with new technology. He is working at a web agency in Bari (Italy) developing a large variety of websites and web applications since 2010 and as part of Github, he tries to contribute to various projects in his spare time.

I want to thank my great friend Linda for giving me the courage to start this project and continue to support me.

Rahul Taiwala started his career as a web developer in college, learning PHP and Adobe Photoshop. He works with open source technologies: Laravel 4 framework, Code Igniter, MySQL, jQuery, and Bootstrap are some of the tools he has up his sleeve. He also works with C# and VB.NET.

In his free time, he loves to work on personal projects. He also does some freelance and consulting work. He knows he has a lot to learn, but his experience has taught him to solve real-world and business problems.

He is currently working as a web developer at MadlyMint Technovative; he is also one of the directors at MadlyMint Technovative.

I would like to thank my family and friends for all their support and encouragement.

www.PacktPub.com

Support files, eBooks, discount offers, and more

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

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

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



<http://PacktLib.PacktPub.com>

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

Why Subscribe?

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

Free Access for Packt account holders

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

Table of Contents

Preface	1
Chapter 1: Welcome to the World of Laravel	7
Landing yourself into the MVC world	8
Why Laravel 4 is awesome! And why it must be your next framework	10
Composer	11
Summary	14
Chapter 2: Let's Begin the Journey	15
Installing Laravel 4 on Windows	16
Installing Composer	16
Installing Laravel 4	16
Installing Laravel 4 on Linux/Ubuntu	17
Installing Laravel 4 on Mac	19
Exploring the Laravel 4 structure	20
Configuring Laravel	23
Configuring the Laravel environment	24
Configuring the database	26
Configuring the application	27
Artisan – magic of Laravel 4	28
Generating a boilerplate controller	29
Managing database with migrations	29
Filling the database with basic data for testing via database seeds	29
Running unit tests	29
Maintenance mode	30
Summary	30
Chapter 3: Creating a Simple CRUD Application in Hours	31
Getting familiar with Laravel 4	32
Controllers versus routes	35
Creating a simple CRUD application with Laravel 4	36
Listing the users – read users from database	37

Creating new users	45
Editing user information	49
Deleting user information	52
Adding pagination to our list users	52
Summary	53
Chapter 4: Building a Real-life Application with Laravel 4 – The Foldagram	55
Preparing the schema	57
Setting up the layout	61
Setting up the inner pages	72
Creating the newsletter section	74
Creating a Foldagram form	77
Summary	81
Chapter 5: Creating a Cart Package for Our Application	83
Introducing IoC container	84
Dependency Injection	84
Service providers	88
Packages in Laravel 4	89
Package structure	90
Facades	93
Cart functions	96
The Cart class	96
Adding Foldagram to the cart	98
Updating the cart	101
Deleting from cart	103
Viewing the cart contents	104
Viewing the cart total	104
Deleting all items from the cart	104
Integrating the Cart package in Foldagram order process	106
Adding the Foldagram information to the Foldagram table	106
Image resizing in Laravel	109
Adding the recipient information to the Recipients table	110
Adding the Foldagram order details to our Cart package	111
Creating the preview page to preview Foldagram	113
Deleting the recipient's information	116
Editing the Foldagram information	116
Deleting Foldagram from the cart	117
Summary	118
Chapter 6: User Management and Payment Gateway Integration	119
Introducing the Sentry package	120
Setting up our user section	121
Register user	123

User login	128
The User dashboard	130
Change password	134
Checkout & payment gateway integration	136
Building the checkout page for credit cards	138
Integrating Stripe payment gateway	141
Creating the checkout order process	142
Building the credits section	143
Building the view orders section	149
Summary	151
Chapter 7: The Admin Section	153
Building the foundation for the administration section	154
Creating a login section for the administrator	158
Managing orders	162
Building the view recipients section	167
Building the order details section	168
Updating order status	170
Deleting orders	173
Exporting orders	174
Managing Foldagram pricing	175
Adding credit for the user	177
Managing users	179
Adding users	182
Editing users	185
Deleting users	185
Blocking users	186
Summary	187
Chapter 8: Building a RESTful API with Laravel – Store Locator	189
REST basics	190
A store locator's single page web application	191
Creating a REST API in Laravel 4 using Resource Controllers	191
Creating a RESTful backend	192
Creating an API to view all the stores	195
Building an API method for viewing an individual store	195
Creating an API method for searching the stores	196
Adding a store method to our API	197
Updating the store method of our API	198
Creating an API method for deleting a store client	199
Creating a frontend via a RESTful API	201
Summary	204

Chapter 9: Optimizing and Securing Our Applications	205
Handling errors	205
Profiling Laravel applications	207
Logging data with Laravel	210
Security in Laravel	210
SQL injections	211
CSRF	211
XSS (Cross site scripting)	212
Summary	213
Chapter 10: Deploying Laravel Applications	215
Creating production configuration	216
Creating a directory structure based on your web host	217
Uploading your Laravel application directory files	217
Deploying via SSH	218
Uploading files via OpenSSH (Linux, Mac)	218
Uploading files via Putty (Windows)	219
Creating a database in the production site and uploading your local database on the production site	219
Giving proper permissions to your storage files	220
Setting up .htaccess based on your server	220
Deploying via FTP	221
Deploying via SSH from the Git repository	221
Deploying via FTP from the Git repository	222
Summary	224
Chapter 11: Creating a Workflow and Useful Laravel Packages and Tools	225
Creating a workflow	226
Introducing JeffreyWay/Laravel-4-Generators	229
Summary	232
Index	233

Preface

This book is about Laravel 4 and its features, and how to use them in real-world projects.

This book will walk you through every single step you need to learn as a beginner to develop Laravel applications with project examples for each. I have used a real-world project that I have developed in Laravel 4 for one of my clients as a sample project to help you go through each process of project development in Laravel 4.

This book covers most of the things you should know when you are developing a project in Laravel 4.

What this book covers

Chapter 1, Welcome to the World of Laravel, introduces the basic MVC concepts and Laravel 4 to you and explains why you should develop your next project in Laravel 4.

Chapter 2, Let's Begin the Journey, explains how to install Laravel 4 and the configuration settings you may need to change.

Chapter 3, Creating a Simple CRUD Application in Hours, will take you through a 360-degree spin ride of how awesome Laravel is for simple CRUD applications and how you can build them with Laravel 4 in hours instead of days.

Chapter 4, Building a Real-life Application with Laravel 4 – The Foldagram, explains how to start a real-world project in Laravel 4. It starts with the requirements. It also explains how you can build migrations in Laravel to manage your database in versions. Then, at the end of the chapter, it explains how to build the frontend of your project as well as some of the frontend features of the project.

Chapter 5, Creating a Cart Package for Our Application, explains how Laravel 4 handles packages. Then, it includes the core concepts of the Laravel framework for us to understand the relationship between packages and the IoC container and learn how we can create packages. Then, at the end, we will see how to build our own Cart package, which can be used in our project as well as other projects.

Chapter 6, User Management and Payment Gateway Integration, explains how to build user registration, login, and dashboard pages for our project. You will learn how to authenticate users and to integrate the cart with the payment gateway. You will also learn how to utilize existing Laravel packages, and how to find them and integrate them in your projects.

Chapter 7, The Admin Section, explains how to build the administration area of our project. You will learn how to build the backend foundation of the project and then the features of our project, such as managing orders, exporting orders, adding credit to users, managing users, resetting passwords, and blocking users.

Chapter 8, Building a RESTful API with Laravel – Store Locator, explains how we can create RESTful applications with Laravel and also how to build an Ajax-powered, one-page store locator application based on the RESTful API built with Laravel.

Chapter 9, Optimizing and Securing Our Applications, explains how you can optimize your applications with the Laravel profiler and how to add security layers in Your application.

Chapter 10, Deploying Laravel Applications, explains how to deploy Laravel applications. It teaches you how to deploy application with FTP or SSH and how to use Git to manage automatic deployments.

Chapter 11, Creating a Workflow and Useful Laravel Packages and Tools, explains some of the useful packages that can be used to create web applications rapidly in Laravel. It also gives some tips on the tools and workflows that will make your life easy as a developer.

What you need for this book

Throughout this book, I have assumed that you have the following programs/packages installed:

- Composer
- PHP 5.3.7 or later
- MCrypt PHP extension
- One of the databases, such as MySQL, MSSQL, and PGSQL

If you don't have them, don't worry. *Chapter 2, Let's Begin the Journey* of this book will guide you through installing them.

Who this book is for

This book is for developers who are new to the Laravel framework as well as developers who want to explore the new breed of Laravel 4, which has been completely redeveloped by Taylor Otwell. You will need to know the basics of PHP and MySQL as well as some basic concepts about object-oriented programming.

If you are tired of dealing with the old way of developing PHP applications and want to use Laravel 4 to manage your project and learn how to do it step-by-step, then this book is for you.

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "You can define RESTful controllers via Laravel 4, and Controllers can have a predefined method to receive a request via GET, PUT, DELETE, POST, and UPDATE."

A block of code is set as follows:

```
$env = $app->detectEnvironment(array(
    'local' => array('your-machine-name'),
    'production' => array('example.com'),
));
```

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

```
return array(
    'connections' => array(
        'mysql' => array(
            'driver' => 'mysql',
            'host' => 'db.example.com',
            'database' => 'myapp',
            'username' => 'myappuser',
            'password' => 'myapppassword',
            'charset' => 'utf8',
```





```
        'collation' => 'utf8_unicode_ci',  
        'prefix'    => '',  
    ),  
    )  
);
```

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

```
# php artisan routes
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Also, we have an **Add User** button which will allow the user to add a new user into the system."

 Warnings or important notes appear in a box like this.

 Tips and tricks appear like this.

Reader feedback

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

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

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

Customer support

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

Downloading the example code

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

Errata

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

Piracy

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

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

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

Questions

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

1

Welcome to the World of Laravel

In this chapter we are going to look at what MVC is and how it fits in web development. We will also look into some of the unique features of the Laravel framework, and we will see later what Laravel 4 offers us and how it is helpful to the developers.

There comes a time in every developer's life where you really want to change the way you work. You want to organize your code; you really care about how you are dividing things and try to refactor your code. You try to learn advanced patterns so that you can manage things, and as you move project-by-project, you develop a framework around your code, which you will use each time you create a new project.

You are beginning to realize there are other options too such as **Frameworks**. Frameworks are developed by the community, and you start to explore them and realize that there are some really good patterns. You also realize that a lot of the grunt work, which you have to do at the time of project initiation, has been done in the framework so that you can enjoy the experience of coding. You think you were stupid enough to not use this in your projects in the first place. You start thinking about objects, classes, patterns, and procedures.

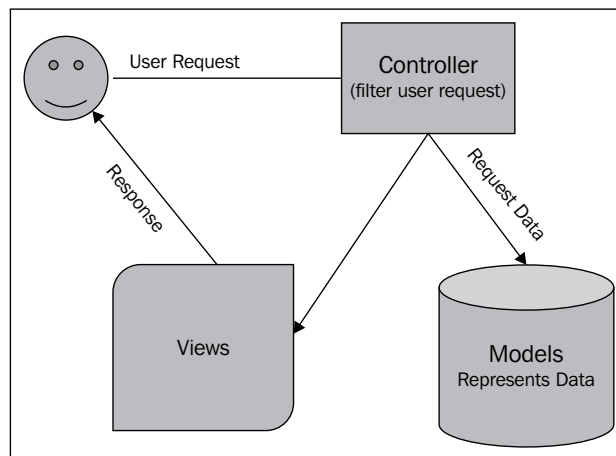
Then it strikes to you, the moment of truth, and you decide to put yourself in the shoes of the framework. This is where this book helps you by guiding you step-by-step through the real-world application development via **Laravel 4 Framework**. I have chosen to walk you carefully through the real-world application development where scopes change, and you have to be ready for constantly changing your code. I will show you how you can effectively use Laravel to minimize the impact of changes and how you can develop applications that take away the pain you feel as a developer when changes constantly evolve in your application.

Landing yourself into the MVC world

When you look around in today's development world, you can find MVC everywhere. Ruby on Rails, ASP.NET, and PHP Frameworks such as CakePHP and Code Igniter all are using it. So what is it that makes MVC such an important part of all these frameworks?

The answer is **Separation of Logic** from your representation layer (Views/HTML). MVC allows you to write the code that can be divided on the basis of three things:

- **Model:** Models are the way by which you can interact with data. It's a layer between your data and your application. Data can be in the database systems such as MySQL or MSSQL or simple Excel or XML files.
- **Views:** Views are the visual representation of your application. Views allow us to write our presentation layer from our business logic.
- **Controller:** Controller is a link between your Model and Views. The primary responsibility of a Controller is to handle requests and pass data from Model to Views.



The way MVC gives you the structure you can actually create separate views for single models, that is, if you are creating an e-commerce site, think of a products page. It can have multiple views such as the Product List View or Product Gallery View. In the MVC world, you will create one model for your product table and via that one model you can generate multiple views.

In a typical MVC web framework, you will find that Controllers handle user request and user data validation and business logic, Models represent data, and Views generate output data. Here is a Laravel framework example of our product's Controller, Model, and View:



Downloading the example code

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

The code example for products Controller is as follows:

```
class ProductsController extends BaseController{

    public function listIndex()
    {
        $products = Products::all();
        return View::make('products', compact('products') );
    }

    public function galleryIndex()
    {
        $products = Products::all();
        return View::make('products_gallery', compact('products') );
    }
}
```

Here, the Products Controller executes the function based on the URI route. For example, if the route is <http://sample.com/list>, it will execute the `listIndex` function which will get all products via the Products Model defined as follows:

The code example for products Model is as follows:

```
Class Products extends Eloquent {}
```

The code example for gallery View is as follows:

```
<ul>
    @foreach ($products as $product)
        <li>{{ $product->name }}</li>
    @endforeach
</ul>
```

The preceding view is for the product gallery view. We are iterating the products array and displaying it on the gallery View:

The code example for list View is as follows:

```
<div>
  @foreach ($products as $product)
    
    <span>{{ $product->name }}</span>
  @endforeach
</div>
```

The preceding view is for the product list view. We are iterating the products array and displaying it on the List view.

Now, if the client asks for changing the order of products in the descending order after six months, all we have to do is change the code we are using to fetch this data, and we will know that we have a fixed Controller for our products. We can then instantly recognize the place where we would have to make the change instead of scanning hundreds of files. If you want to change the look of site, you can just change Views. So, MVC provides us a structure that is easily recognizable as well as we know that when we change things, it will not affect other portions of our web pages.

Why Laravel 4 is awesome! And why it must be your next framework

The philosophy of Laravel's creator *Taylor Otwell* about Laravel is that:

Development should be a creative experience that you enjoy, not something that is painful.

The framework reflects that Laravel is a simple and elegant framework for PHP web development with expressive and artistic syntaxes. But there is more to Laravel 4. It is one of the few frameworks that have implemented **Composer** and in such a way that it gives you full control over what you want to choose in your framework. Let's review what has changed in the PHP community and how Laravel has adopted these changes in its significant Version 4.

The PHP community has been going through a revolution over the last few years. Those of you who are still not aware of the recent PHP development should read the guide *phptherightway* (<http://www.phptherightway.com/>) on what has changed in the recent PHP versions and how the community is creating an echo system that will change the way PHP is used in projects. One important thing that happened during these revolution is that a lot of framework developers met at php|tek in 2009 and formed the **FIG (Framework Interoperability Group)**. The group formed standards that allow them to work together and share code even if frameworks are different. These standards are called **PSR-0**, **PSR-1**, **PSR-2**, and **PSR-3**. PSR- standards lay the foundation for **Composer**, which is a dependency management system for PHP. To know more about PSR standards, visit <http://net.tutsplus.com/tutorials/php/psr-huh/>.

So what is dependency? Dependency is a class or package we use in our code to achieve some specific tasks such as sending e-mails via PHPMailer or Swift Mailer. Now in this case, PHPMailer and Swift Mailer is a dependency of our application. Remember that a dependency can have another dependency, that is, Swift Mailer could be using some other third-party class or package that you are not aware of.

Composer

Composer is a dependency management tool for PHP. The most important thing about composer is that it can manage the dependency of your project dependencies; that is, if one of the libraries you are using in your project is dependent on two other libraries, you don't have to manually find and update anything to upgrade them. All three libraries can be updated via a single command, that is, Composer Update. Composer can manage dependency up to *N* level means all dependency of your project via single tool which is really powerful option to have when you are using lot of libraries. The other benefit is that it manages the auto load file, which will include all your dependencies. So you don't have to remember all paths to your dependencies and include each of them on every file of your project, just include the autoload file provided by composer.

So what is it that makes Laravel 4 stand out from other frameworks? We will discuss these features as follows:

- **Composer ready:** Composer is the way the PHP community is going and there are thousands (7,000 currently) of packages already available in the Composer package archive (<https://packagist.org/>). Laravel 4 was designed in such a way that it integrates Composer packages easily. Laravel 4 itself uses some of the Composer components such as Symfony2's HTTP package for routing in Laravel 4. Say you want to send an SMTP e-mail with attachments. Download a Swift Mailer component with Composer, and you can directly use the Swift Mailer component anywhere in your project.

- **Interoperability:** Laravel 4 is actually a mixture of different Composer components. All of these components are available in Github and all are 100 percent unit tested. Some of those components are carefully chosen by *Taylor Otwell* from very carefully crafted frameworks such as Symphony 2.
- Laravel 4 gives you the power of choosing what you like. For example, you don't like the Mail component of Laravel, which is Swift Mailer actually, and you want to replace it with the **PHPMailer** component which you really like and is also Composer ready; thus, it will be a very easy operation to switch these two. You can swipe components as and when you need via the Composer and Laravel configuration.
- You don't have to wait for months for updating that one feature you want for your database component. You can just simply update individual components via the composer update.
- You can further develop core components yourself and use it in your projects without having to worry about the impact of changes in your framework.

Here are some of the unique features of Laravel:

- **Eloquent ORM:** Laravel has one of the most advanced built-in ORMs called **Eloquent** that allows you to write database tables as objects and also allows you to interact with those objects. You can actually perform database operations without writing SQL queries. And it's also kind of unique because it allows you to maintain a relationship between tables, so you can perform lot of operations on multiple tables without writing long queries via objects.
- **Routes:** Developers can write their logic either in Routers or Controllers. This can be handful when you have a small site, or you want to define static pages quickly. You would not need to write Controllers for every page of your site.
- **RESTful:** Laravel 4 provides us with a unique way of creating RESTful APIs. You can define RESTful controllers via Laravel 4, and Controllers can have predefined methods to receive requests via GET, PUT, DELETE, POST, and UPDATE. We will see this in later chapters in detail when we build RESTful APIs with the help of Laravel 4.
- **Auto loading:** Laravel 4 automatically loads all your dependencies such as libraries, packages, and Models. You don't have to manually load anything. Laravel 4 is smart about anything you use in your code since it will automatically load that class or library when you use it.
- **IOC containers:** An IOC container is Laravel's way to manage independent components and provide a unique way to control them via a single API.

- **Migrations:** Most of PHP frameworks take migrations, which is version management for your database. Laravel 4 has introduced migration with DB Seeds, which allows you to manage your database version as well as add some seed data as and when you need.
- **Pagination:** Laravel 4 makes pagination breeze by, giving you a powerful library that allows you to write your pagination in simply three lines, which in other frameworks or a manual code would take hundreds of line of the code. We will see this in later chapters in detail when we build our application.
- **Unit testing:** Laravel Framework was built with unit testing in mind. The framework itself provides unit tests for all of its components, and that's one of the reasons it's very stable as well as fast paced as constant changes can be tested via unit tests. You can also add your unit tests directly just like you write application classes. It allows various features such as seeding database from unit test classes and custom session handling, so you can test pretty much everything for your application.
- **Artisan:** Artisan is the command-line tool of Laravel that allows you to perform a lot of grunt tasks you would hate to do manually. Artisan can be used to create your basic Controller, Model, and View files directly via the command line. It can also build migrations which can be handy to manage your database schema. With Laravel 4, you can create your own commands and do useful stuff such as send pending mails to people, or repair database, or anything that may be required occasionally for your application. You can also create database seeds that will allow you to seed some data initially. Artisan can also run unit tests for your application. We will see all of this in detail during our application development.
- **Queues:** Laravel 4 has this really nice feature that every web developer dreams of – Queues. Yes, you can queue your tasks in Laravel 4 and what's so amazing about it is that users don't have to wait anymore for very long tasks; like you are generating multipage PDF or sending a lot of e-mails. You can just queue your tasks, and it will get executed later without slowing down the process.
- **Events:** Laravel 4 provides events and you can hook your code into application-specific Laravel events. It's one of the features that really help developers. You can also create custom events for your application, or you can hook into Laravel 4 application-specific events such as `Laravel.log`, `Laravel.query` event.

I believe Laravel's features make a huge difference in how easy it is to write reusable, maintainable code. It is one of the major aspects of any framework. Laravel provides us with a base platform, which removes us from all cumbersome tasks and gives us a clean syntax. It also includes awesome tools to develop our project in such a way that we can enjoy the whole experience of doing it as a developer. Laravel 4 is the fresh air the PHP community needed after years of stalled development. Laravel reassembles new PHP 5.3 features and gives us many more features as a framework, which previously was not possible. And that's all about Laravel 4 features. I have just given the basic features in later chapters. We will see in detail the various features of Laravel 4 and explore how you can actually use them in your projects.

Summary

So we have learned how the MVC world works, what a composer is, and how it helps you in managing the dependencies of your application, and what Laravel 4 offers as an MVC framework and its unique features. In the next chapter let's dig deeper into the world of Laravel 4. If this excites you, let's begin the journey of Laravel 4.

2

Let's Begin the Journey

In this chapter, we are going to learn how we can install and configure Laravel 4. We will see how Laravel structures itself. Also, we'll talk about some useful configurations you should know before starting your project in Laravel 4.

Let's start our journey. The first step of our journey is to install Laravel successfully. Laravel 4 has few system requirements:

- Composer
- PHP 5.3.7 or later
- MCrypt PHP extension

We will see how to install Laravel 4 on the following operating systems:

- Windows 7
- Ubuntu Linux
- Mac OS X

In this chapter, we will see how we can install and configure Laravel 4 on our development environment. We will also learn the basic foundations of the Laravel 4 structure, learn where our code resides, and how Laravel 4 manages dependencies and gives us a simple syntax while hiding complex stuff.

Installing Laravel 4 on Windows

You will need to install a web server on Windows. If you don't want to fight your way around by installing different components, fixing path errors, and manually editing a lot of files, you can download a prepackaged development environment that has everything configured in one single bundle. Here are some of the popular environments for Windows:

- WAMP (<http://www.wampserver.com/en/>)
- XAMPP (<http://www.apachefriends.org/en/xampp.html>)

Installing Composer

Once you have successfully installed a development environment, you will need to install Composer. Composer could be installed either locally or globally. If you want to install Composer globally, download the Composer setup from the Composer website (<http://www.getcomposer.org/download>). If you want to install Composer in your project directory, you can download it via executing the following command on the Windows command line:

```
> php -r "eval('?'>'.file_get_contents('https://getcomposer.org/installer'));"
```

It will download the `composer.phar` file into your project directory. You can test Composer by executing the following command:

```
> php composer.phar about
```

If you have installed it globally, you don't need to use PHP, you can run the following command:

```
> composer about
```

If Composer is installed correctly, it will give you the output `Composer-package management for PHP`.

Installing Laravel 4

Now, as Composer is installed, let's install Laravel 4 by downloading the Laravel 4 ZIP file from <https://github.com/laravel/laravel/archive/master.zip>; alternatively, you can even install it using the Composer command by going to the root of your server (generally, `C:\xampp\www` in case of XAMPP or `C:\wamp\www` for a WAMP server).

```
> composer create-project laravel/laravel <myproject>
```

The preceding command will install and configure Laravel for you in the `myproject` directory at the root of your web server.

If you have downloaded the ZIP file, extract it in your web server's `www` folder. So if you have installed XAMPP in `C:\xampp`, generally the path would be `C:\xampp\www`; if WAMP, it would be `C:\WAMP\`. You will have the `Laravel` folder in your XAMPP folder. Now the `Laravel` directory will be there, but it's not ready for use. You need to download packages before you can run Laravel 4. The way this works is via Composer; you can download all the packages required to install and run Laravel 4 after the initial barebone files. Laravel 4 comes with the `composer.json` file that has all the dependent packages in `.json` format, which Composer can understand and download. The great thing about Composer is that it can download dependent packages of the package you are downloading. Now all you have to do to install packages is run the following command on the command line from the root directory of your project.

If you have Composer installed globally:

```
> composer install
```

If you have Composer installed in the `Project` directory:

```
> php composer.phar install
```

What this command will do is install all the dependencies mentioned in `composer.json` into the `Vendors` directory located at the root directory of your project. Also Composer creates an autoload file in the `Vendor` directory that can be used by any framework to autoload packages as and when needed.

So now you have Laravel 4 installed. Go visit your `http://localhost/laravel` or `http://127.0.0.1/laravel` page. You might wonder what happened, why does it show directory listing. Well, you need to go to the public directory, so actually it would be either `http://localhost/laravel/public` or `http://127.0.0.1/laravel/public`. If all goes well, you will be greeted by the **You have Arrived** page. If you get this far, congratulations! You have successfully installed Laravel 4 in your Windows development environment; we will see how to further configure our environment later in the chapter.

Installing Laravel 4 on Linux/Ubuntu

You will need to install a web server before installing anything else. You can do that by installing a LAMP server directly using the following command:

```
$ sudo tasksel
```

If the preceding command says `taskset` is not found, you would need to install `taskset` using the following command:

```
$ sudo apt-get install taskset
```

`Taskset` will show you different packages that you can install; select the LAMP server from the list and install the package. Once you install it, you can install `Composer` using the following command:

```
$ curl -sS https://getcomposer.org/installer | php
```

Or if you don't have `curl` installed, use the following command:

```
$ php -r "eval('?'>'.file_get_contents('https://getcomposer.org/installer'))';"
```

This will install `Composer` into the local directory of your project. To install it globally, you can copy your `composer.phar` file to your `bin` directory in `/usr/local/`.

Now you can download `Laravel 4` from the download section at <http://laravel.com> or <https://github.com/laravel/laravel/archive/master.zip>; once the download is complete, you can extract it into `/var/www` of your system, which is generally the document root of your LAMP server.

After extracting the files, you will need to download `Laravel` packages via `Composer`; here is command for doing that:

```
$ php Composer.phar install
```

One more step you need to make sure for `Laravel 4` to work is that of setting up appropriate permissions. You need to give correct permissions to the storage folder of the `Laravel` framework. You can do this using the following command:

```
$ sudo chmod -R o+w storage
```

So now you have `Laravel 4` Installed. Go visit your <http://localhost/laravel> or <http://127.0.0.1/laravel> page. You might wonder what happened, why does it show directory listing. Well, you need to go to the public directory, so actually it would be either <http://localhost/laravel> or <http://127.0.0.1/laravel/public>. If all goes well, you will be greeted by the **Hello world** page. If you get this far, congratulations! You have successfully installed `Laravel 4` in your Linux Ubuntu development environment. We will see how to further configure our environment later in this chapter.

Installing Laravel 4 on Mac

You will need to install a web server before installing anything else. You can do that by installing the MAMP server using the following prepackaged development environment:

- MAMP stack (<http://www.mamp.info/en/mamp/index.html>)

Once you install the MAMP server, you can install Composer using the following command line:

```
$ brew tap josegonzalez/homebrew-php
$ brew install josegonzalez/php/composer
```

Or if you don't have homebrew installed, use the following command:

```
$ php -r "eval('?'>'.file_get_contents('https://getcomposer.org/
installer'));"
```

This will install Composer into the local directory of your project. To install it globally, you can copy your `composer.phar` file to your `bin` directory in `/usr/local/`.

Now you can download Laravel 4 from the download section at <http://laravel.com> or <https://github.com/laravel/laravel/archive/master.zip>; once the download is complete, you can extract it into `/var/www` of your system, which is generally the document root of your MAMP server.

After extracting the files, you will need to download Laravel packages via Composer; here is the command for doing that:

```
$ php Composer.phar install
```

One more step you need to make sure for Laravel 4 to work is that of setting up appropriate permissions. You need to give correct permissions to the storage folder of the Laravel framework. Here is the command for that:

```
$ sudo chmod -R o+w storage
```

So now you have Laravel 4 installed. Go visit your <http://localhost/laravel> or <http://127.0.0.1/laravel> page. You might wonder what happened, why does it show directory listing. Well, you need to go to the public directory, so actually it would be either <http://localhost/laravel/public> or <http://127.0.0.1/laravel/public>. If all goes well, you will be greeted by the **Hello world** page. If you get this far, congratulations! You have successfully installed Laravel 4 in your Mac development environment. We will see how to further configure our environment later in this chapter.

Exploring the Laravel 4 structure

Now as you have installed Laravel 4 successfully, let's explore what's inside Laravel 4, how it functions, and what are its core components.

Name	Size	Type
▼ app	12 items	folder
▶ commands	0 items	folder
▶ config	12 items	folder
▶ controllers	2 items	folder
▶ database	3 items	folder
▶ lang	1 item	folder
▶ models	1 item	folder
▶ start	3 items	folder
▶ storage	5 items	folder
▶ tests	2 items	folder
▶ views	2 items	folder
filters.php	2.1 kB	PHP script
routes.php	455 bytes	PHP script
▶ bootstrap	4 items	folder
▼ public	4 items	folder
▶ packages	0 items	folder
favicon.ico	0 bytes	Microsoft icon
index.php	2.0 kB	PHP script
robots.txt	25 bytes	plain text document
▶ vendor	16 items	folder
artisan	2.4 kB	PHP script
composer.json	493 bytes	plain text document
composer.lock	61.0 kB	plain text document
phpunit.xml	566 bytes	XML document
server.php	519 bytes	PHP script

As we can see in preceding screenshot, Laravel 4 has four directories and four files in the root directory. Here is a breakdown of each of them:

Directory/File	Usage
/app	This directory contains all your controllers, models, and views. In short, all the logic needed to run your application written by you would be available here.
/bootstrap	This directory contains all the files such as <code>start.php</code> , <code>paths.php</code> , and <code>autoload.php</code> that bootstrap the Laravel 4 application. Any request that hits your application will go through <code>start.php</code> first.
/public	This directory is the public part of your site. Here you'll find <code>index.php</code> , which actually bootstraps an instance of the Laravel 4 app. All your assets will also go here.
/vendor	This directory contains third-party packages that Laravel 4 uses. As we already know from <i>Chapter 1, Welcome to the World of Laravel</i> , Laravel 4 is made up of individual components, and those components can be found here.
artisan	This is Laravel's magic command-line tool that will make your life better as a developer. We will look at <code>artisan</code> briefly later in this chapter.
composer.json	The <code>composer.json</code> directory is a file with all your project package dependencies along with their version numbers.
phpunit.xml	The <code>phpunit.xml</code> directory is a support file for testing with PHPUnit testing tools.
server.php	This file provides Apache's <code>mod_rewrite</code> facility for a built-in PHP server testing without installing a web server.

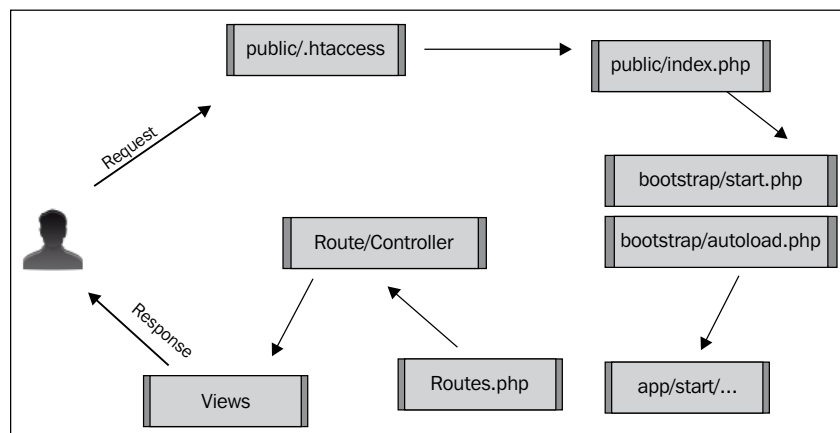
We will be working extensively on the `/app` directory of our project. So let's get a closer look at what it contains and how we can develop our app around this directory.

The following table shows Laravel 4's app directory structure in detail:

Directory/File	Usage
/commands	This directory allows you to create your own custom artisan commands. We will see this later in detail.
/config	This directory contains the entire configuration your project may need, which includes your environment information, that is, either the local, staging, or live environment. It also contains database settings, session settings, and many more configurations you can think of.
/controllers	This directory allows you to create a controller class that provides a gateway to models and views.

Directory/File	Usage
/database	This directory contains migrations that actually build your database via version management and database seeds, which can be used to create runtime data to test your application.
/lang	This directory provides you with a way to make your application multilingual. The lines for pagination and form validation messages are in English.
/models	Models deal with all your data. Model classes provide a simple way of managing data and will contain all your business logic.
/start	The start directory has settings for your custom artisan commands as well as all global and local settings.
/storage	The storage directory is where Laravel stores its cache and session files. Laravel needs read/write permission on this directory for this reason.
/tests	The tests directory provides you with a simple location where you can write all your unit tests, and on top of this, you can run those tests via Laravel's artisan command line.
/views	This is your theme directory or the presentation layer. You can write all your HTML templates here with template engines such as twig or blade.
Filters.php	This file allows you to catch any request before it hits the route or controller, and you can process it for reasons such as security or authentication.
Routes.php	This file has all the routing information that helps Laravel to transfer a request to either route or controller.

Now we know a lot of things about Laravel; let's see how a typical user request flows through the entire Laravel 4 framework and how response gets generated.



The preceding diagram shows how a request is handled between different parts of the framework; here is a brief overview of what happens during different phases:

1. When a user sends a page request such as `/index` or `/about`, it first hits the `.htaccess` file in the public directory of the Laravel framework. The `.htaccess` file ensures that each request hits the `index.php` file, which is essential as only one entry point allows us to set all types of configurations and filter requests.
2. The `index.php` file first loads `bootstrap/autoload.php`, which loads Composer packages including `vendor/autoload.php`. It also loads Laravel's custom class loader.
3. Next, `index.php` loads `bootstrap/start.php`, and an application object is created. This invokes the creation of a request object.
4. Next, Laravel loads the environment and configurations, and it invokes the router object that checks which route/controller will be used for the request.
5. Based on route/controller response, an object is created and returned to the user.

Now we have some understanding of how Laravel 4 works. Let's look at some essential configurations.

Configuring Laravel

As we already know, all of our configurations are stored at `/app/config`. The way Laravel handles configuration is by storing each configuration in the key-value store as an array. This is a very flexible way of storing configurations as we can rewrite them as and when needed.

Laravel manages different types of configurations in different types of files. These are as follows:

- `app.php`: This file contains time zone settings, language settings, and debug settings.
- `auth.php`: This file contains the basic authentication settings.
- `database.php`: This file contains settings for databases. It will have information such as the username, password, and URL for where we could connect.

Here is an example of the workbench configuration file that Laravel provides:

```
<?php

return array(

    'name' => '',

    'email' => '',

);
```

The preceding configuration is used by Laravel when we generate a new package via the artisan `workbench` command. We can get this configuration value anywhere in our project via `option`. An example of this is as follows:

```
$option = Config::get('workbench.name');
```

We can even set the configuration as follows:

```
$option = Config::set('workbench.name', 'Taylor');
```

On top of this, Laravel also allows us to set our own configuration files. You can create your own configuration file in `/app/config/`. You can load the configuration using the preceding command, but instead of `workbench`, use the filename without an extension and key from the `config` file; for example:

```
$option = Config::get('custom.name');
```

Configuring the Laravel environment

One of the most painful aspects that a lot of web developers have is of managing multiple environments, such as their local development environment, prestaging/dev environment, staging environment, and live environment. Often developers have to struggle a lot to manage each environment and its configurations, such as database settings, debug settings, and a lot of these settings.

Laravel makes it very easy to manage multiple environments in your project. You can define environments within the `config` directory. For each environment, you can create an environment-specific directory and store configuration files such as database, session, or the mail settings file, which will override your default settings provided in `/app/config/` for that environment. But you will be wondering how Laravel recognizes which environment to use.

Laravel makes it very simple for us. It reads the `/bootstrap/start.php` file and loads the environment array to determine which environment to load. See the following example:

```
$env = $app->detectEnvironment(array(
    'local' => array('computername'),
    'dev' => array('http://85.205.120.11'),
    'Staging' => array('http://stage.oursite.com'),
    'Live' => array('http://oursite.com'),
));
```

Here as you can see, 'local' is the environment name, and you need to create that directory in your `/app/conf/` directory. Within that directory, you can specify all your settings for a local environment. Laravel will try to match the `computername` or `http` host and use an environment that matches the value pair given in this environment and load its settings.

Here one thing to remember is that we just need to provide the key-value pairs that we actually want to override. Laravel will load another configuration from the default file provided in default configuration files.

Refer to the following screenshot on how you can manage different environments in your project:

Name	Size	Type	Date Modified
commands	0 items	folder	Tuesday 14 May 2013 11:10:22 AM IST
config	15 items	folder	Wednesday 22 May 2013 04:14:42 PM IST
live	1 item	folder	Wednesday 22 May 2013 04:40:11 PM IST
database.php	3.3 kB	PHP script	Tuesday 14 May 2013 11:10:22 AM IST
local	1 item	folder	Wednesday 22 May 2013 04:40:12 PM IST
database.php	3.3 kB	PHP script	Tuesday 14 May 2013 11:10:22 AM IST
packages	0 items	folder	Tuesday 14 May 2013 11:10:22 AM IST
staging	1 item	folder	Wednesday 22 May 2013 04:40:15 PM IST
database.php	3.3 kB	PHP script	Tuesday 14 May 2013 11:10:22 AM IST
testing	2 items	folder	Tuesday 14 May 2013 11:10:22 AM IST
app.php	7.3 kB	PHP script	Tuesday 14 May 2013 11:10:22 AM IST
auth.php	1.9 kB	PHP script	Tuesday 14 May 2013 11:10:22 AM IST

If you noticed, you may find that two directories are already available in the default Laravel framework files:

- testing
- packages

The `testing` directory contains unit-testing-specific configurations that run when you perform unit testing via the artisan command line. The `packages` directory will contain all package-specific configurations that you need to override from the given package. You can even define an environment-specific package configuration file that overrides default package configurations provided in the package by placing it in `/app/config/packages/vendor/package/environment`.

You can also detect which environment you are in your code using the following line of code:

```
$environment = App::environment();
```

You can use this information for things such as checking whether we are in the development environment and whether to send an e-mail or not based on the environment.

Now let's look at some default configurations you may need to edit, such as a database or an application.

Configuring the database

The default database configuration in Laravel 4 is stored in the `database.php` file placed at `app/config`. Here you can create all of your database-connection-related configurations. Laravel 4 supports the following databases: MySQL, PostgreSQL, SQLite, and SQL server. You can create your environment-specific file in `/app/config/environmentname/database.php`.

Here are some settings you need to configure for your database in `app/config/database.php`. The first setting you need to configure is which database you are going to use in your project. You can change that using the following line of code:

```
'default' => 'mysql',
```

Here `default` is used to indicate which connection to use from the connections array you specify in the `database.php` file. You are telling Laravel 4 to use `mysql` settings provided in the connections array. Here is how connections array works:

```
'connections' => array(

    'log_sqlite' => array(
        'driver'     => 'sqlite',
        'database'   => __DIR__.'/../database/production.sqlite',
        'prefix'     => '',
    ),
    'result_sqlite' => array(
        'driver'     => 'sqlite',
```

```

        'database' => __DIR__.'../database/production.sqlite',
        'prefix'   => '',
    ),

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

```

In the preceding example, you can see we have defined all the connections we need to use in our project; of course, you can add other connections as well. So if you are using a SQL server, the config file already provides you with all the settings in the `sqlsrv` key. Remember that the default database provided in the default key will be used when you are querying the database via the `DB` class or `Eloquent`. To use another connection, you need to specify which database to use during querying or in the `Eloquent` object. Here is a sample database query which will load SQLite database and query it:

```
$users = DB::connection('log_sqlite')->select('select * from log');
```

Or if you are using `Eloquent`:

```

class log extends Eloquent {
    public static $connection = 'log_sqlite';
}

```

So this is how you can manage multiple connections via the configuration file and Laravel 4.

Configuring the application

Application-related configurations are stored in `app/config/app.php`. We will see how they affect our application. Here is brief list of configurations in `app.php` file:

- **Debug:** This is used for getting a detailed error message with a stack trace. You should always turn off this when you are going live with your application. To turn off you can set it as `False`.
- **URL:** This should be the root of your application. It is used by the artisan command-line tool for artisan tasks, that is, `dev.test.com`.

- **Time zone:** It is used for setting the time zone. This will be used when running the date/time functions by Laravel as a base time zone, that is, in UTC.
- **Locale:** This is used for setting the language locale you are going to use in your application. For example, you can set it as *es* for Spanish. You would need to create a language directory at `/app/lang/es` and specify the language files.
- **Key:** This is used by Laravel 4 to encrypt things such as passwords and cookies. It should be a long and random key. By default, Laravel will generate the key for us. At times when you deploy or change the session or cookie values, you can generate it via the artisan command; we will discuss this in detail in the next chapter.
- **Providers:** This is for the list of service providers you want to autoload. If you see the list, this is how Laravel 4 autoloads all the components.

Artisan – magic of Laravel 4

Artisan is the command-line tool provided in Laravel 4. It does a lot of painful tasks that web developers would have previously had to do manually. Here is a list of tasks Laravel 4 can help you with:

- Generating a boilerplate controller
- Managing database with migrations
- Filling the database with basic data for testing via database seeds
- Running unit tests
- Routing
- Configuring an application
- Creating new artisan commands
- Creating a new workbench package

Artisan resides at the root of our framework directory. You would need to use command line to access artisan commands. So open your command directory and change the directory to where you have extracted the framework and run the following command. Say you have installed the web server at `/var/www` and copied Laravel at `/var/www/l4`, then execute the following command:

```
$ cd /var/www/l4
$ php artisan list
```

The preceding command will show a list of available artisan commands.

Generating a boilerplate controller

To generate a boilerplate controller along with all the methods, execute the following command:

```
$ php artisan controller make:user
```

This will generate the controller at `/app/controller/user.php` and `user.php`. It will have the entire code you will basically need to create a controller with all the methods for a boilerplate. Later in *Chapter 3, Creating a simple CRUD Application in Hours*, we will see how it really helps us in developing apps quickly.

Managing database with migrations

Migration is a version management tool for database. It allows you to create a schema for each of your table, and a new migration file is created for each change that is made to your database. We will see migrations in detail in later chapters. Laravel provides the following commands to manage migrations:

Command	Description
\$ php artisan migrate:install	Creates the migration repository
\$ php artisan migrate:make	Creates a new migration file
\$ php artisan migrate:refresh	Resets and reruns all the migrations
\$ php artisan migrate:reset	Rollback all the database migrations
\$ php artisan migrate:rollback	Rollback the last database migration

Filling the database with basic data for testing via database seeds

Artisan allows us to feed data with the database seeds file in the `app/database/seeds` directory. It will execute all the database seeds when we run the following command:

```
$ PHP artisan db:seed
```

Running unit tests

Artisan allows us to run unit tests defined in the `app/tests` directory. The test class should extend `TestCase`. You can use the methods just as PHPUnit. To run unit tests, you would need to execute the following command:

```
$ php artisan test
```

Maintenance mode

One of the new features of Laravel 4 is that we can now switch on or off the maintenance mode. This really helps when you are upgrading or maintaining your application. To enable the maintenance mode, simply execute the `down` artisan command as follows:

```
$ php artisan down
```

To disable the maintenance mode, use the `up` command as follows:

```
$ php artisan up
```

Summary

So we have learned how Laravel can be installed in different environments, Laravel's structure, and lots of configurations that will come handy when we start our project.

In later chapters, we will see the following artisan commands in detail:

- Creating new artisan commands
- Creating a new workbench package

In the next chapter, we will learn how to create CRUD applications with Laravel.

3

Creating a Simple CRUD Application in Hours

Now that we know the basics of Laravel 4, let's start working our way into our first simple CRUD application. We will see how Controllers and routes work and how we can create views. Then we will build our first CRUD application in hours and not in days via some artisan-powered magic. Here are the topics we are going to cover in this chapter.

- Getting familiar with Laravel 4
- Creating a simple CRUD application with Laravel 4
 - Listing the users
 - Creating new users
 - Editing user information
 - Deleting user information
 - Adding pagination to our list users

Getting familiar with Laravel 4

I assume you have followed the instructions provided in *Chapter 2, Let's Begin the Journey*, and installed Laravel 4. Now if everything is installed correctly you will be greeted by this beautiful screen, as shown in the following screenshot, when you hit your browser with `http://localhost/laravel/public` or `http://localhost/<installeddirectory>/public`:



Now that you can see we have installed Laravel correctly, you would be thinking how can I use Laravel? How do I create apps with Laravel? Or you might be wondering why and how this screen is shown to us? What's behind the scenes? How Laravel 4 sets this screen for us? So let's review that.

When you visit the `http://localhost/laravel/public`, Laravel 4 detects that you are requesting for the default route which is `/`. You would be wondering what route is this if you are not familiar with the MVC world. Let me explain that.

In traditional web applications we use a URL with page name, say for example:

`http://www.shop.com/products.php`

The preceding URL will be bound to the page `products.php` in the web server hosting `shop.com`. We can assume that it displays all the products from the database. Now say for example, we want to display a category of books from all the products. You will say, "Hey, it's easy!" Just add the category ID into the URL as follows:

`http://www.shop.com/products.php?cat=1`

Then put the filter in the page `products.php` that will check whether the category ID is passed. This sounds perfect, but what about pagination and other categories? Soon clients will ask you to change one of your category page layouts to change and you will hack your code more. And your application URLs will look like the following:

- `http://www.shop.com/products.php?cat=2`
- `http://www.shop.com/products.php?cat=3&page=1&total=20`
- `http://www.shop.com/products.php?cat=3&page=1&total=20&layout=1`

If you look at your code after six months, you would be looking at one huge `products.php` page with all of your business and view code mixed in one large file. You wouldn't remember those easy hacks you did in order to manage client requests. On top of that, a client or client's SEO executive might ask you why are all the URLs so badly formatted? Why are they are not human friendly? In a way they are right. Your URLs are not as pretty as the following:

- `http://www.shop.com/products`
- `http://www.shop.com/products/books`
- `http://www.shop.com/products/cloths`

The preceding URLs are human friendly. Users can easily change categories themselves. In addition to that, your client's SEO executives will love you for those URLs just as a search engine likes those URLs.

You might be puzzled now; how do you do that? Here my friend **MVC (Model View Controller)** comes into the picture. MVC frameworks are meant specifically for doing this. It's one of the core goals of using the MVC framework in web development.

So let's go back to our topic "routing"; routing means decoupling your URL request and assigning it to some specific action via your controller/route. In the Laravel MVC world, you register all your routes in a route file and assign an action to them. All your routes are generally found at `/app/routes.php`.

If you open your newly downloaded Laravel installation's `routes.php` file, you will notice the following code:

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

The preceding code registers a route with `/` means default URL with view `/app/views/hello.php`. Here view is just an `.html` file. Generally view files are used for managing your presentation logic. So check `/app/views/hello.php`, or better let's create an about page for our application ourselves.

Let's register a route about by adding the following code to `app/routes.php`:

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

We would need to create a view at `app/views/about.php`. So create the file and insert the following code in to it:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>About my little app</title>
</head>
<body>
  <h1>Hello Laravel 4!</h1>
  <p> Welcome to the Awesomeness! </p>
</body>
</html>
```

Now head over to your browser and run `http://localhost/laravel/public/about`. You will be greeted with the following output:

Hello Laravel 4!

Welcome to the Awesomeness!

Isn't it easy? You can define your route and separate the view for each type of request. Now you might be thinking what about Controllers as the term MVC has C for Controllers? And isn't it difficult to create routes and views for each action? What advantage will we have if we use the preceding pattern? Well we found that mapping URLs to a particular action in comparison to the traditional one-file-based method. Well first you are organizing your code way better as you will have actions responding to specific URLs mapped in the route file.

Any developer can recognize routes and see what's going on with your code. Developers do not have to check many files to see which files are using which code. Your presentation logic is separated, so if a designer wants to change something, he will know he needs to look at the `view` folder of your application.

Now about Controllers; they allow us to group related actions into a single class. So in a typical MVC project, there will be one user Controller that will be responsible for all user-related actions, such as registering, logging in, editing a profile, and changing the password. Generally routes are used for small applications or creating static pages quickly. Controllers provide more in-depth options to create a group of methods that belong to a specific class related to the application.

Here is how we can create Controllers in Laravel 4. Open your `app/routes.php` file and add following code:

```
Route::get('contact', 'Pages@contact');
```

The preceding code will register the `http://yourapp.com/contact` URL in the Pages Controller's `contact` method. So let's write a page's Controller. Create a file `PagesController.php` at `/app/controllers/` in your Laravel 4 installation directory. The following are the contents of the `PagesController.php` file:

```
<?php
class PagesController extends BaseController {

    public function contact()
    {
        return View::make('hello');
    }

}
```

Here `BaseController` is a class provided by Laravel so we can place our Controller shared logic in a common class. And it extends the framework's Controller class and provides the Controller functionality. You can check `Basecontroller.php` in the Controller's directory to add shared logic.

Controllers versus routes

So you are wondering now, "What's the difference between Controllers and routes?" Which one to use? Controllers or routes? Here are the differences between Controllers and routes:

- A disadvantage of routes is that you can't share code between routes, as routes work via `Closure` functions. And the scope of a function is bound within function.
- Controllers give a structure to your code. You can define your system in well-grouped classes, which are divided in such a way that it makes sense, for example, users, dashboard, products, and so on.
- Compared to routes, Controllers have only one disadvantage and it's that you have to create a file for each Controller; however, if you think in terms of organizing the code in a large application, it makes more sense to use Controllers.

Creating a simple CRUD application with Laravel 4

Now as we have a basic understanding of how we can create pages, let's create a simple CRUD application with Laravel 4. The application we want to create will manage the users of our application. We will create the following list of features for our application:

- List users (read users from the database)
- Create new users
- Edit user information
- Delete user information
- Adding pagination to the list of users

Now to start off with things, we would need to set up a database. So if you have phpMyAdmin installed with your local web server setup, head over to <http://localhost/phpmyadmin>; if you don't have phpMyAdmin installed, use the MySQL admin tool workbench to connect with your database and create a new database.

Now we need to configure Laravel 4 to connect with our database. So head over to your Laravel 4 application folder, open `/app/config/database.php`, change the MySQL array, and match your current database settings. Here is the MySQL database array from `database.php` file:

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

Now we are ready to work with the database in our application. Let's first create the database table `Users` via the following SQL queries from phpMyAdmin or any MySQL database admin tool;

```
CREATE TABLE IF NOT EXISTS 'users' (
    'id' int(10) unsigned NOT NULL AUTO_INCREMENT,
    'username' varchar(255) COLLATE utf8_unicode_ci NOT NULL,
```

```
'password' varchar(255) COLLATE utf8_unicode_ci NOT NULL,
'email' varchar(255) COLLATE utf8_unicode_ci NOT NULL,
'phone' varchar(255) COLLATE utf8_unicode_ci NOT NULL,
'name' varchar(255) COLLATE utf8_unicode_ci NOT NULL,
'created_at' timestamp NOT NULL DEFAULT '0000-00-00 00:00:00',
'updated_at' timestamp NOT NULL DEFAULT '0000-00-00 00:00:00',
PRIMARY KEY ('id')
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci AUTO_
INCREMENT=3 ;
```

Now let's seed some data into the Users table so when we fetch the users we won't get empty results. Run the following queries into your database admin tool:

```
INSERT INTO 'users' ('id', 'username', 'password', 'email', 'phone',
'name', 'created_at', 'updated_at') VALUES
(1, 'john', 'johndoe', 'johndoe@gmail.com', '123456', 'John', '2013-
06-07 08:13:28', '2013-06-07 08:13:28'),
(2, 'amy', 'amy.deg', 'amy@outlook.com', '1234567', 'amy', '2013-06-07
08:14:49', '2013-06-07 08:14:49');
```



Later we will see how we can manage our database via Laravel 4's powerful migrations features. At the end of this chapter, I will introduce you to why it's not a good practice to manually create SQL queries and make changes to the database structure. And I know that passwords should not be plain too!

Listing the users – read users from database

Let's read users from the database. We would need to follow the steps described to read users from database:

- A route that will lead to our page
- A controller that will handle our method
- The Eloquent Model that will connect to the database
- A view that will display our records in the template

So let's create our route at `/app/routes.php`. Add the following line to the `routes.php` file:

```
Route::resource('users', 'UserController');
```

If you have noticed previously, we had `Route::get` for displaying our page Controller. But now we are using `resource`. So what's the difference?

In general we face two types of requests during web projects: GET and POST. We generally use these HTTP request types to manipulate our pages, that is, you will check whether the page has any POST variables set; if not, you will display the user form to enter data. As a user submits the form, it will send a POST request as we generally define the `<form method="post">` tag in our pages. Now based on page's request type, we set the code to perform actions such as inserting user data into our database or filtering records.

What Laravel provides us is that we can simply tap into either a GET or POST request via routes and send it to the appropriate method. Here is an example for that:

```
Route::get('/register', 'UserController@showUserRegistration');
Route::post('/register', 'UserController@saveUser');
```

See the difference here is we are registering the same URL, `/register`, but we are defining its GET method so Laravel can call `UserController` class' `showUserRegistration` method. If it's the POST method, Laravel should call the `saveUser` method of the `UserController` class.

You might be wondering what's the benefit of it? Well six months later if you want to know how something's happening in your app, you can just check out the `routes.php` file and guess which Controller and which method of Controller handles the part you are interested in, developing it further or solving some bug. Even some other developer who is not used to your project will be able to understand how things work and can easily help move your project. This is because he would be able to somewhat understand the structure of your application by checking `routes.php`.

Now imagine the routes you will need for editing, deleting, or displaying a user. Resource Controller will save you from this trouble. A single line of route will map multiple restful actions with our resource Controller. It will automatically map the following actions with HTTP verbs:

HTTP VERB	ACTION
GET	READ
POST	CREATE
PUT	UPDATE
DELETE	DELETE



To know more about REST actions refer to https://blog.apigee.com/detail/restful_api_design.

On top of that you can actually generate your Controller via a simple command-line artisan using the following command:

```
$ php artisan Usercontroller:make users
```

This will generate `UserController.php` with all the RESTful empty methods, so you will have an empty structure to play with. Here is what we will have after the preceding command:

```
class UserController extends BaseController {

    /**
     * Display a listing of the resource.
     *
     * @return Response
     */
    public function index()
    {
        //
    }

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

    /**
     * Store a newly created resource in storage.
     *
     * @return Response
     */
    public function store()
    {
        //
    }

    /**
     * Display the specified resource.
     *
     * @param int $id
     */
}
```

```
    * @return Response
    */
    public function show($id)
    {
        //
    }

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

    /**
     * Update the specified resource in storage.
     *
     * @param int $id
     * @return Response
     */
    public function update($id)
    {
        //
    }

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

Now let's try to understand what our single line route declaration created relationship with our generated Controller.

HTTP VERB	Path	Controller Action/method
GET	/Users	Index
GET	/Users/create	Create
POST	/Users	Store
GET	/Users/{id}	Show (individual record)
GET	/Users/{id}/edit	Edit
PUT	/Users/{id}	Update
DELETE	/Users/{id}	Destroy

As you can see, resource Controller really makes your work easy. You don't have to create lots of routes. Also Laravel 4's artisan-command-line generator can generate resourceful Controllers, so you will write very less boilerplate code. And you can also use the following command to view the list of all the routes in your project from the root of your project, launching command line:

```
$ php artisan routes
```

Now let's get back to our basic task, that is, reading users. Well now we know that we have `UserController.php` at `/app/controller` with the `index` method, which will be executed when somebody launches `http://localhost/laravel/public/users`. So let's edit the Controller file to fetch data from the database.

Well as you might remember, we will need a Model to do that. But how do we define one and what's the use of Models? You might be wondering, can't we just run the queries? Well Laravel does support queries through the DB class, but Laravel also has Eloquent that gives us our table as a database object, and what's great about object is that we can play around with its methods. So let's create a Model.

If you check your path `/app/models/User.php`, you will already have a user Model defined. It's there because Laravel provides us with some basic user authentication. Generally you can create your Model using the following code:

```
class User extends Eloquent {}
```

Now in your controller you can fetch the user object using the following code:

```
$users = User::all();
$users->toArray();
```

Yeah! It's that simple. No database connection! No queries! Isn't it magic? It's the simplicity of Eloquent objects that many people like in Laravel.

But you have the following questions, right?

- How does Model know which table to fetch?
- How does Controller know what is a user?
- How does the fetching of user records work? We don't have all the methods in the `User` class, so how did it work?

Well models in Laravel use a lowercase, plural name of the class as the table name unless another name is explicitly specified. So in our case, `User` was converted to a lowercase `user` and used as a table to bind with the `User` class.

Models are automatically loaded by Laravel, so you don't have to include the reference of the Model file. Each Model inherits an Eloquent instance that resolves methods defined in the `model.php` file at `vendor/Laravel/framework/src/Illuminate/Database/Eloquent/` like `all`, `insert`, `update`, `delete` and our user class inherit those methods and as a result of this, we can fetch records via `User::all()`.

So now let's try to fetch users from our database via the Eloquent object. I am updating the `index` method in our `app/controllers/UsersController.php` as it's the method responsible as per the REST convention we are using via resource Controller.

```
public function index()
{
    $users = User::all();

    return View::make('users.index', compact('users'));
}
```

Now let's look at the `view` part. Before that, we need to know about Blade. Blade is a templating engine provided by Laravel. Blade has a very simple syntax, and you can determine most of the Blade expressions within your view files as they begin with `@`. To print anything with Blade, you can use the `{{ $var }}` syntax. Its PHP-equivalent syntax would be:

```
<?php echo $var; ?>
```

Now back to our view; first of all, we need to create a view file at `/app/views/users/index.blade.php`, as our statement would return the view file from `users.index`. We are passing a compact `users` array to this view. So here is our `index.blade.php` file:

```
@section('main')

<h1>All Users</h1>

<p>{{ link_to_route('users.create', 'Add new user') }}</p>

@if ($users->count())
    <table class="table table-striped table-bordered">
```

```

        <thead>
            <tr>
                <th>Username</th>
                <th>Password</th>
                <th>Email</th>
                <th>Phone</th>
                <th>Name</th>
            </tr>
        </thead>

        <tbody>
            @foreach ($users as $user)
                <tr>
                    <td>{{ $user->username }}</td>
                    <td>{{ $user->password }}</td>
                    <td>{{ $user->email }}</td>
                    <td>{{ $user->phone }}</td>
                    <td>{{ $user->name }}</td>
                    <td>{{ link_to_route('users.edit', 'Edit',
array($user->id), array('class' => 'btn btn-info')) }}</td>
                    <td>
                        {{ Form::open(array('method'
=> 'DELETE', 'route' => array('users.destroy', $user->id))) }}
                        {{ Form::submit('Delete', array('class' =>
'btn btn-danger')) }}
                        {{ Form::close() }}
                    </td>
                </tr>
            @endforeach

        </tbody>

    </table>
@else
    There are no users
@endif

@stop

```

Let's see the code line by line. In the first line we are extending the user layouts via the Blade template syntax `@extends`. What actually happens here is that Laravel will load the layout file at `/app/views/layouts/user.blade.php` first.

Here is our `user.blade.php` file's code:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <link href="//netdna.bootstrapcdn.com/twitter-bootstrap/2.3.1/
css/bootstrap-combined.min.css" rel="stylesheet">
    <style>
      table form { margin-bottom: 0; }
      form ul { margin-left: 0; list-style: none; }
      .error { color: red; font-style: italic; }
      body { padding-top: 20px; }
    </style>
  </head>

  <body>

    <div class="container">
      @if (Session::has('message'))
        <div class="flash alert">
          <p>{{ Session::get('message') }}</p>
        </div>
      @endif

      @yield('main')
    </div>

  </body>

</html>
```

Now in this file we are loading the Twitter bootstrap framework for styling our page, and via `yield('main')` we can load the main section from the view that is loaded. So here when we load `http://localhost/laravel/public/users`, Laravel will first load the `users.blade.php` layout view and then the main section will be loaded from `index.blade.php`.

Now when we get back to our `index.blade.php`, we have the main section defined as `@section('main')`, which will be used by Laravel to load it into our layout file. This section will be merged into the layout file where we have put the `@yield('main')` section.

We are using Laravel's `link_to_route` method to link to our route, that is, `/users/create`. This helper will generate an HTML link with the correct URL. In the next step, we are looping through all the user records and displaying it simply in a tabular format. Now if you have followed everything, you will be greeted by the following screen:

All Users

[Add new user](#)

Username	Password	Email	Phone	Name		
John	johndoe	johndoe@gmail.com	123456	John	Edit	Delete
amy	amy.deg	amy@outlook.com	78546546	amy	Edit	Delete

Creating new users

Now as we have listed our users, let's write the code for creating new users. To create a new user, we will need to create the Controller method and bind the view for displaying the new user form. We would need to create the Controller method for saving the user too. Now as we have bound the resourceful Controller, we don't need to create separate routes for each of our request. Laravel will handle that part if we use REST methods.

So first let's edit the controller at `/app/controllers/UsersController.php` to add a method for displaying the view:

```
public function create()
{
    return View::make('users.create');
}
```

This will call a view at `/app/views/users/create.blade.php`. So let's define our `create.blade.php` view as follows:

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

@section('main')

<h1>Create User</h1>

{{ Form::open(array('route' => 'users.store')) }}
<ul>

    <li>
```

```
        {{ Form::label('name', 'Name:') }}
        {{ Form::text('name') }}
    </li>

    <li>
        {{ Form::label('username', 'Username:') }}
        {{ Form::text('username') }}
    </li>

    <li>
        {{ Form::label('password', 'Password:') }}
        {{ Form::password('password') }}
    </li>

    <li>
        {{ Form::label('password', 'Confirm Password:') }}
        {{ Form::password('password_confirmation') }}
    </li>

    <li>
        {{ Form::label('email', 'Email:') }}
        {{ Form::text('email') }}
    </li>

    <li>
        {{ Form::label('phone', 'Phone:') }}
        {{ Form::text('phone') }}
    </li>

    <li>
        {{ Form::submit('Submit', array('class' => 'btn')) }}
    </li>
</ul>
{{ Form::close() }}

@if ($errors->any())
    <ul>
        {{ implode('', $errors->all('<li class="error">:message</li>')) }}
    </ul>
@endif

@stop
```

Let's try to understand our preceding view. Here we are extending the users layout we created in our `List Users` section. Now in the main section, we are using Laravel's `Form` helper to generate our form. This helper generates HTML code via its methods such as `label`, `text`, and `submit`.

Refer to the following code:

```
{{ Form::open(array('route' => 'users.store')) }}
```

The preceding code will generate the following HTML code:

```
<form method="POST" action="http://localhost/users"
accept-charset="UTF-8">
```

As you can see it's really convenient for us to not worry about linking things correctly. Now let's create our `store` method to store our form data into our users table:

```
public function store()
{
    $input = Input::all();
    $validation = Validator::make($input, User::$rules);

    if ($validation->passes())
    {
        User::create($input);

        return Redirect::route('users.index');
    }

    return Redirect::route('users.create')
        ->withInput()
        ->withErrors($validation)
        ->with('message', 'There were validation errors.');
```

Here we are first validating all the input that came from the user. The `Input::all()` function fetches all the `$_GET` and `$_POST` variables and puts it into a single array. The reason why we are creating the single input array is so we can check that array against validation rules' array. Laravel provides a very simple `Validation` class that can be used to check validations. We could use it to check whether validations provided in the rules array are followed by the input array by using the following line of code:

```
$validation = Validator::make ($input, User::$rules);
```

Rules can be defined in an array with validation attributes separated by the column "|". Here we are using `User::$rules` where `User` is our Model and it will have following code:

```
class User extends Eloquent {

    protected $guarded = array('id');
    protected $fillable = array('name', 'email');

    public static $rules = array(
        'name' => 'required|min:5',
        'email' => 'required|email'
    );
}
```

As you can observe we have defined two rules mainly for name and e-mail input fields. If you are wondering about `$guarded` and `$fillable` variables, these variables are used to prevent mass assignment. When you pass an array into your Model's `create` and `update` methods, Laravel tries to match the right columns and sets values in the database. Now for instance, if a malicious user sends a hidden input named `id` and changes his ID via the `update` method of your form, it could be a huge security hole; to prevent this, we should define the `$guarded` and `$fillable` arrays. The `$guarded` array will guard the columns defined in the guarded array, that is, it will prevent anyone from changing values in that column. The `$fillable` array will only allow elements defined in `$fillable` to be updated.

Now we can use the `$validation` instance we created to check for validations.

```
$result = $validation->passes();
echo $result; // True or false
```

If you see our code now, we are checking for validation via the `passes()` method in our `Store()` method of `UserController`. Now if validation gets passed, we can use our user Model to store data into our database. All you need to do is call the `Create` method of the Model class with the `$input` array. So refer to the following code:

```
User::create($input);
```

The preceding code will store our `$input` array into the database; yes, it's equivalent to your SQL query.

```
Insert into user(name,password,email,city) values (x,x,...,x);
```

Here we have to fill either the `$fillable` or `$guarded` array in the model, otherwise, Laravel will throw a mass assignment exception. Laravel's Eloquent object automatically matches our input array with the database and creates a query based on our input array. Don't you think this is a simple way to store input into the database? If user data is inserted, we are using Laravel's `redirect` method to redirect it to our list of users' pages. If validation fails, we are sending all of the input with errors from the validation object into our create users form.

Editing user information

Now as we learned how easy it is to add and list users. Let's jump into editing a user. In our list users view we have the edit link with the following code:

```
{{ link_to_route('users.edit', 'Edit', array($user->id), array('class'
=> 'btn btn-info')) }}
```

Here, the `link_to_route` function will generate a link `/users/<id>/edit`, which will call the resourceful Controller `user`, and Controller will bind it with the `edit` method.

So here is the code for editing a user. First of all we are handling the edit request by adding the following code to our `UserController`:

```
public function edit($id)
{
    $user = User::find($id);
    if (is_null($user))
    {
        return Redirect::route('users.index');
    }
    return View::make('users.edit', compact('user'));
}
```

So when the edit request is fired, it will hit the `edit` method described in the preceding code snippet. We would need to find whether the user exists in the database. So we use our user model to query the ID using the following line of code:

```
$user = User::find($id);
```

Eloquent object's `find` method will query the database just like a normal SQL.

```
Select * from users where id = $id
```

Then we will check whether the object we received is empty or not. If it is empty, we would just redirect the user to our list user's interface. If it is not empty, we would direct the user to the user's edit view with our Eloquent object as a compact array.

So let's create our edit user view at `app/views/users/edit.blade.php`, as follows:

```
@extends('users.scaffold')

@section('main')

<h1>Edit User</h1>
{{ Form::model($user, array('method' => 'PATCH', 'route' =>
array('users.update', $user->id))) }}
<ul>
    <li>
        {{ Form::label('username', 'Username:') }}
        {{ Form::text('username') }}
    </li>
    <li>
        {{ Form::label('password', 'Password:') }}
        {{ Form::text('password') }}
    </li>
    <li>
        {{ Form::label('email', 'Email:') }}
        {{ Form::text('email') }}
    </li>
    <li>
        {{ Form::label('phone', 'Phone:') }}
        {{ Form::text('phone') }}
    </li>
    <li>
        {{ Form::label('name', 'Name:') }}
        {{ Form::text('name') }}
    </li>
    <li>
        {{ Form::submit('Update', array('class' => 'btn btn-
info')) }}
        {{ link_to_route('users.show', 'Cancel', $user->id,
array('class' => 'btn')) }}
    </li>
</ul>
{{ Form::close() }}

@if ($errors->any())
    <ul>
        {{ implode('', $errors->all('<li class="error">:message</
li>')) }}
    </ul>
@endif

@stop
```

Here we are extending our users' layout as always and defining the main section. Now in the main section, we are using the `Form` helper to generate a proper REST request for our controller.

```
{{ Form::model($user, array('method' => 'PATCH', 'route' =>
array('users.update', $user->id))) }}
```

Now, you may have not dealt with the method `PATCH` as we only know of two protocols, `GET` and `POST`, as most browsers generally support only these two methods. The REST method for editing is `PATCH`, and what Laravel does is that it creates a hidden token so it knows which method to call. So the preceding code generates the following code:

```
<form method="POST" action="http://ch3.sr/users/1" accept-
charset="UTF-8">
<input name="_method" type="hidden" value="PATCH">
```

It actually fires a `POST` method for browsers that are not capable for handling the `PATCH` method. Now, when a user submits this form, it will send a request to the update method of `UserController` via the resourceful Controller we set in `routes.php`.

Here is the update method of `UserController`:

```
public function update($id)
{
    $input = Input::all();
    $validation = Validator::make($input, User::$rules);
    if ($validation->passes())
    {
        $user = User::find($id);
        $user->update($input);
        return Redirect::route('users.show', $id);
    }
    return Redirect::route('users.edit', $id)
        ->withInput()
        ->withErrors($validation)
        ->with('message', 'There were validation errors.');
```

Here Laravel will pass the ID of the user we are editing in the update method. We can use this ID to find the user via our user model's Eloquent object's `find` method. Then we will update the Eloquent object with an input array just like we did in the insert operation.

Deleting user information

To delete a user we can use the `destroy` method. If you go to our user lists view, you can find the following delete link's generation code:

```
{{ Form::open(array('method' => 'DELETE', 'route' => array('users.  
destroy', $user->id))) }}  
{{ Form::submit('Delete', array('class' => 'btn btn-danger')) }}  
{{ Form::close() }}
```

The preceding code is handled by Laravel similar to the way in which it handles the `PATCH` method. Laravel will generate a post request with the hidden method token set as `PATCH`, which it can recognize when it hits the Laravel request object.

At `UserController` this request will hit the `destroy()` method as follows:

```
public function destroy($id)  
{  
    User::find($id)->delete();  
    return Redirect::route('users.index');  
}
```

Laravel will directly send `id` to the `destroy` method, so all we have to do is use our user model and delete the record with its `delete` method. If you noticed Eloquent allows us to chain methods. Isn't it sweet? So there would be no queries but just one line to delete a user.

That's one of the reasons to use Eloquent objects in your projects. It allows you to quickly interact with the database and you can use objects to match your business logic.

Adding pagination to our list users

One of the painful tasks most developers face often is that of pagination. With Laravel it's no more the case, as Laravel provides a simple approach to set pagination to your pages.

Let's try to implement pagination to our list user's method. To set pagination we can use the `paginate` method with Laravel's Eloquent object. Here is how we can do that:

```
public function index()  
{  
    $users = User::paginate(5);  
    return View::make('users.index', compact('users'));  
}
```

The preceding code is the `index` method of the `UsersController` class, which we were using previously for getting all the users with `User::all()`. We just used the `paginate` method to find five records from the database. Now in our list users view, we can use the following code to display pagination links:

```
{{ echo $users->links(); }}
```

Here the `links()` method will generate pagination links for you. And best part, Laravel will manage the code for pagination. So all you have to do is use `paginate` method with your eloquent object and `links` method to display generated links.

Summary

So we have set up our simple CRUD application and now we know how easy it is with Laravel to set up CRUD operations. We have seen Eloquent Laravel's database ORM that makes working with a database simple and easy. We have learned how to list users, create new users, edit users, delete users, and how to add pagination to our application. In the next chapter let's dive into a more complicated application.

4

Building a Real-life Application with Laravel 4 – The Foldagram

One of our friends Jordan needs a web application for his startup `thefoldagram.com`. Foldagram is all about delivering awesome pictures with messages from anywhere in the world to your family and friends in a nicely crafted Foldagram via mail, that too without any of the hassle of managing visits to a mail office in a foreign country. Just create a Foldagram online, upload your picture, type your message, and then preview and send it instantly. Foldagram will deliver your mail via its own mail system.

So I met him and did some basic analysis of what he wants in `thefoldagram.com`. I used mix agile techniques to define the requirements. Here are some of the requirements in terms of stories that we need to make `thefoldagram.com` work.

As a user, I should be able to do the following:

- Create Foldagram from any page on the site
- Preview Foldagram to see how it will look before sending it
- View my cart easily at any time
- Pay for my order via my credit card
- View my order status
- View my past orders
- Purchase credits
- Subscribe to the e-mail newsletter

As an admin, I should be able to do the following:

- Manage orders
- Export orders which are paid but are in the queue for printing
- Add credits to the users
- Manage users
- Reset user passwords
- Block a user

Agile development requires constant iteration and testing, getting a minimum viable product out in the open for users to start using, allowing you to take in their feedback and iterate further.

Here, as a part of our first iteration, we will work on the frontend sprint. We will try to finish the following things in the frontend of our site:

- Preparing the schema
- Setting up the Home page
- Setting up other pages
- Creating a Foldagram form
- Creating a Newsletter section

Now, during each sprint, we create mockups of a page layout before starting any code. If a client approves that mockup, we will go ahead with the coding. That mockup will give the client a certain idea about how the site will look like after the current sprint. Generally, mockups can be built on paper or via software such as **Pencil** (<http://pencil.evolus.vn/>). There are two main advantages of mockups. It gives a designer a rough idea on how to design certain elements of the pages. It also helps in dealing with the changes that generally come when the clients don't have mockups. You will develop a lot of stuff, but the clients will ask for a lot of small changes as they have no idea how it will turn out.

Preparing the schema

So let's prepare a database for our current sprint, meanwhile the designer will have designs for our first sprint. Laravel 4 provides us with migrations to manage our database. **Migrations** are like version management for your database. For each change in your database, you will prepare a migration file which will hold specifically what we are changing in our database. It will also store what should happen if we want to revoke that database change.

This is particularly useful in agile projects as we constantly iterate, and changes are part of a project. As we try to manage our project in sprints, we can deploy database changes quickly for the staging/live servers on the client site or demo area. This eliminates manual fiddling with the database, and you will be saved by that one field which you tend to miss often in deployment. This is because the migration files hold the information of all the changes. Laravel manages migration files so that you can revert or apply changes to your project instantly.

Let's see how we can use migrations. To create a new migration file, you can use an artisan command from your project root, which will generate a migration file with the correct class information:

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

This will generate a migration file `2013_07_19_222248_create_user_table.php` (filename will depend on table name you will give in command, or the `--table` parameter and current date time) at `app\database\migrations\`. Now if you look at the file, you will see the following code:

```
<?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->timestamps();
    });
}

    public function down()
    {
        Schema::drop('users');
    }

}
```

Here, the up function will be used when you run the migration and the down function will be used when you rollback your database. Laravel provides us the Schema class which manages all database related work such as creating or removing tables and adding fields to the existing table. In the previous example, as you can see we are creating the users table via `Schema::create`. Laravel puts sensible defaults such as making the ID field as the primary key, auto increment via the `increments` method, adding the timestamp fields such as `created_at`, and adding `updated_at` via the `timestamps()` method. Similarly, `Schema::drop` will remove the users table from the database.

You can add columns in the `Schema::create` method if you want. So, let's add our user table columns into it:

```
    public function up()
    {
        Schema::create('users', function(Blueprint $table) {
            $table->increments('id');
            $table->string('username', '12');
            $table->string('password', '16');
            $table->string('email', '40');
            $table->string('phone', '12')->nullable();
            $table->string('name', '40');
            $table->boolean('blocked')->default(0);
            $table->timestamps();
        });
    }
}
```

Now, we can run the `artisan` command:

```
$ php artisan migrate
```

This will run all migration files and create tables or other database changes specified in the migration files. In our case, this command will create the `users` table with the columns which we specified. But what if we want to rollback our migration? Well we can rollback our migration with the following command:

```
$ php artisan migrate:rollback
```

Now in our case, the preceding command will drop the `users` table from the database, as we have specified in `down` method.

You might think it's lot of work. Why can't we just set up the database tables directly via **MySQL** or **PHPMyAdmin**? Well, imagine you have multiple members on team and all have different tables which are required for that code. Then how do you manage the databases? Or imagine you have already deployed your project and you are working on second version of the project. You can't remember each change you make in database. So, it's better to create migration files and run them when you deploy your code in production.

Laravel also provides a `Seeder` class, which allows us to seed data into our tables. this becomes very important as you can insert data into tables whenever you want to test your application, or when you don't have backend and you want to start with having some sort of data into your tables.

Laravel comes with a `DatabaseSeeder` class defined at `app/database/seeds/DatabaseSeeder.php`. You can use this class to run your seeds. Here's an example for our `users` table.

```
class DatabaseSeeder extends Seeder {
    public function run()
    {
        $this->call('UserTableSeeder');
        $this->command->info('User' table seeded!);
    }
}
class UserTableSeeder extends Seeder {
    public function run()
    {
```



```
$hashed = Hash::make('secret');

DB::table('users')->insert(
    array('username'=>'James','email' =>
        'james@gmail.com','password'=>$hashed),
    array('username'=>'Steve','email' =>
        'steve@yahoo.com','password'=>$hashed)
);
}
```

To run the previous seed file, we need to run following artisan command:

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

This will run our seed file and insert two records into the database. If you note carefully, we are using `Hash::make` to encrypt our password. Here are the other table schemas we will need for this sprint.

Subscribe the table schema that we will need, so that the users can subscribe to the site:

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

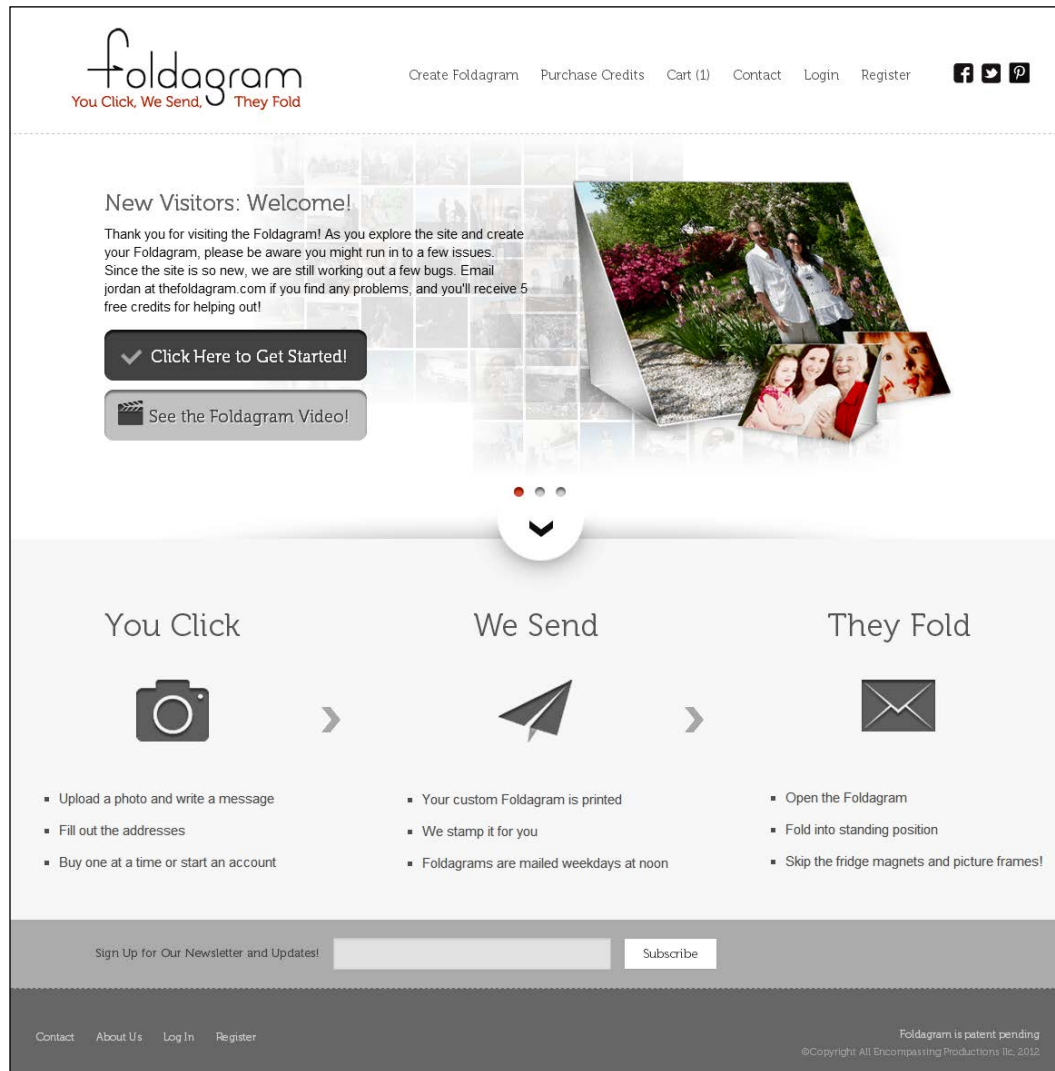
Foldagram table schema for storing Foldagram's created by the user is as follows:

```
Schema::create('foldagram', function(Blueprint $table) {
    $table->increments('id');
    $table->text('message');
    $table->string('image');
    $table->boolean('status');
    $table->integer('user_id');

    $table->integer('exported');
    $table->timestamps();
});
```

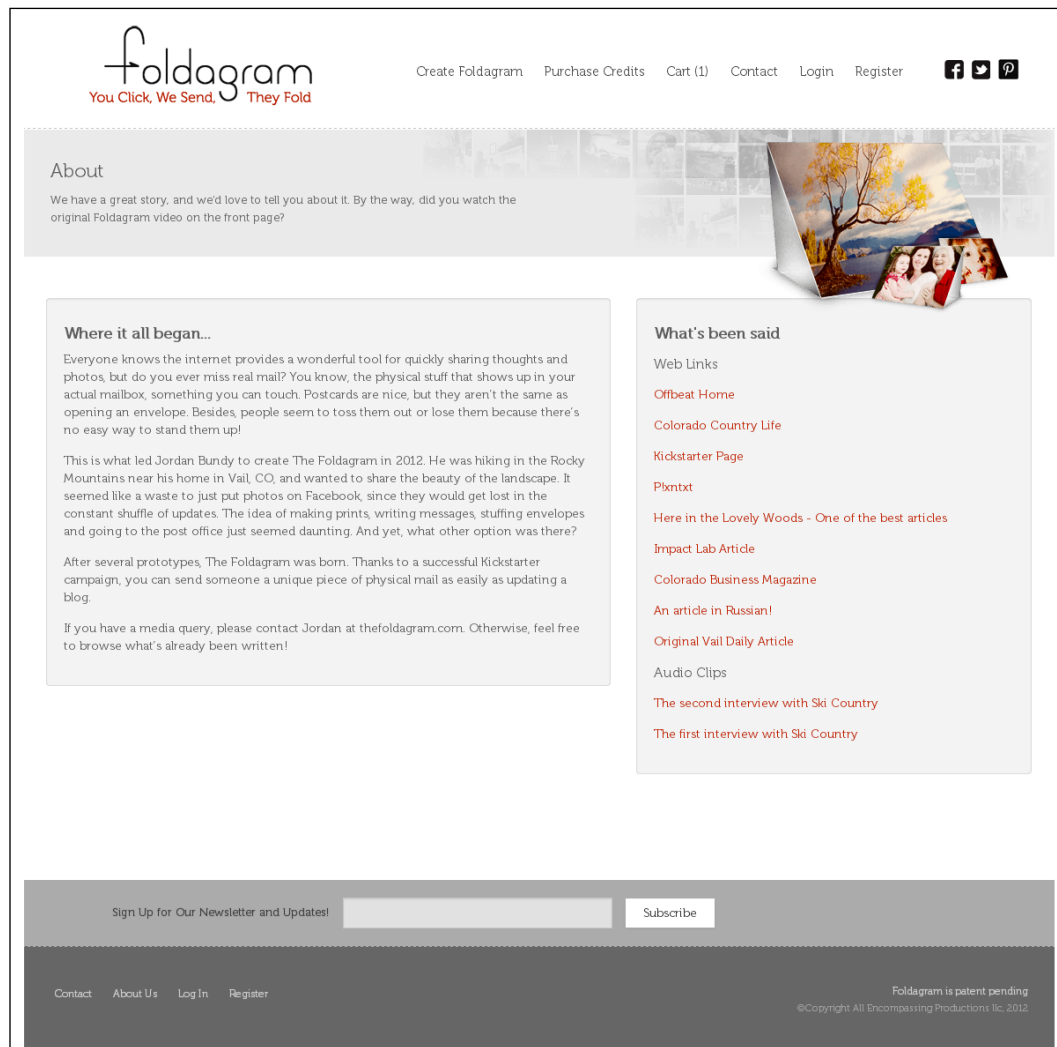
Setting up the layout

As we have completed the schema for this sprint, we now have designs available which are approved by the client. So, we can start working on setting up the layout. But first let's look at the design so we can decide how we can work out our layouts.



Home page of thefoldagram.com

Here is our Home page design. We would need to divide this Home page into Blade sections.



Inner page layout

Here, from the screenshots *Home page of Foldagram.com* and *Inner page layout* we can see that we have an almost similar header and footer in both types of pages, except for a top banner in the inner pages. We will achieve layouts via the Blade Templating engine provided by Laravel. We will create the layout templates for the Home page and the inner page via Blade, and then we will mix the dynamic portion of the templates via sections and yield functionality from the Blade Templating engine.

So let's start by creating layouts for our application. Firstly, we will create a layout for our Home page as it's different from all other pages. I prefer to store all layouts in the layout directory under the views directory. So, I will create `default.blade.php` under `app\views\layouts` for our Home page. Here is the code for the Home page layout:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
  <title>{{ $title }}</title>
  <meta name="viewport" content="width=device-width">
  <link href="//fonts.googleapis.com/css?family=Droid+Sans:400,700"
rel="stylesheet" type="text/css">

  {{ HTML::style('css/bootstrap.css') }}
  {{ HTML::style('css/style.css') }}
  {{ HTML::style('css/slider.css') }}
  {{ HTML::script('js/slider.js') }}s

</head>
<body class = "{{ $class }}">
```

This is the header part of template. You will notice that we are using `HTML::style` and `HTML::script` of the HTML helper to get the CSS and JavaScript files from our `/public` directory. You should use them so you don't have to manually edit the paths if the project location changes or when you upload your application and make it live.

```
<div class="container">
  <div class="row-fluid header">
    <div class="span4 logo">
      <a href="{{ URL::to('/') }}"></a>
    </div>
    <div class="span6 menu">
      <ul>
        <li><a href="#popup" data-toggle="modal">Create
          Foldagram</a></li>
        <li>{{ link_to_route('pcredit', 'Purchase Credits')
          }}</li>
        <li>{{ link_to_route('cart', 'Cart') }}</li>
        <li>{{ link_to_route('contact', 'Contact') }}</li>
        <li>{{ link_to_route('userlogin', 'Login') }}</li>
```

```
        <li>{{ link_to_route('register', 'Register') }}</li>
    </ul>
</div>
<div class="span2 social">
    <a href="https://www.facebook.com/TheFoldagram"
      target="_blank"></a>
    <a href="https://twitter.com/thefoldagram"
      target="_blank"></a>
    <a href="https://pinterest.com/thefoldagram/"
      target="_blank"></a>
</div>
</div>
```

The preceding block starts with our main container which holds everything in our template. You will notice that we are using the URL helper function `URL::Asset` to manage the paths of our `public` directory. If you are setting up a template, always remember not to use absolute paths. Use the Laravel helper functions to manage paths. So, when you want to move location you don't have to worry about paths. For managing our top navigation, we are using the `link_to_route` helper. This manages navigation links by putting the correct path of the router, and we don't have to manually put the paths of our pages. This is again an elegant solution for managing paths, as we will always have routes for each of our pages:

```
@yield('inner-banner')

<div class="row-fluid content">
    @yield('content')
</div>
```

Here, `yield` is used so the other page can override this section. `inneri-banner` is the dynamic section for the inner pages. It's not necessary to have that section for pages which extends the main layout. `content` will be the main section and main content of the page.

```
<div class="row-fluid subscribe-form">
    <div class="span12 subscribe-content">

        {{ Form::open(array('url' => 'subscribe')); }}
    </div>
</div>
```

```

    {{ Form::label('something', 'Sign' Up for Our Newsletter and
Updates!'); }}
    {{ Form::text('email', null, array('class' => 'input-large',
'placeholder' => ')); }}
    {{ Form::submit('Subscribe'); }}
    {{ Form::close(); }}

</div>
</div>

```

The preceding code is used for adding the subscribe e-mail section on every page of the site.

```

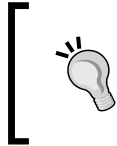
<div class="row-fluid footer">
    <div class="span8 footer-menu">
        <ul>
            <li>{{ link_to_route('contact', 'Contact')
            }}</li>
            <li>{{ link_to_route('about', 'About Us')
            }}</li>
            <li>{{ link_to_route('login', 'Log In')
            }}</a></li>
            <li>{{ link_to_route('register', 'Register')
            }}</a></li>
        </ul>
    </div>
    <div class="span4 copyright">
        <h4>Foldagram is patent pending</h4>
        <p>&copy; Copyright All Encompassing Productions
        llc, 2012</p>
    </div>

</div>
</div><!-- End Container -->

</body>
</html>

```

And the footer section is the last, with the footer navigation and text.



The `{{ $class }}` used in the body tag is something you can use to add a class, which allows you to add custom CSS based on page. Generally, you will pass the class with other data via route or controller when you are calling the view.

`URL::to('/')` will return the URL of the Home page of our application. Similarly, the `link_to_route()` helper method will generate the correct URL to the route that we pass as an argument.

Now, the main part of template is where `@yield('content')` is written. At this point, the content section from the other view will be rendered. Let's see how exactly this process works. First, we will create view for our Home page. Then, we will create the content section and extend the main template. So, our Home view will be at `app/views/home.blade.php`. Here it is:

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

@section('content')
    <div class="row-fluid slider-content">
        <div class="span12 slider">
            <div class="flexslider">
                <ul class="slides">
                    <li class="da-slider">
                        <div class="da-slide row">
                            <div class="span5">
                                <h2>New Visitors: Welcome!</h2>
                                <p>Thank you for visiting the Foldagram! As
                                    you explore the site and create your
                                    Foldagram,
                                </p>
                                <a href="#" popup" data-toggle="modal"
                                    class="da-link create">Click Here to Get
                                    Started!</a>
                                <a href="#" video" class="da-link video"
                                    data-toggle="modal">See the Foldagram
                                    Video!</a>
                            </div>
                            <div class="da-img span7"></div>
                        </div>
                    </li>
                </ul>
            </div>
        </div>
    </div>
```

```

        </div>
    </li>
    <li>
        <div class="da-slide row">
            <div class="span5">
                <h2>Send from anywhere on Earth.</h2>
                <p>Wether you're travelling in South America,
                    China, or your own neighborhood, Foldagrams
                    are always</p>
                <a href="#"popup" data-toggle="modal"
                    class="da-link create">"Click Here to Get
                    Started!</a>
                <a href="#"video" class="da-link video"
                    data-toggle="modal">"See the Foldagram
                    Video!</a>
            </div>
            <div class="da-img span7"></div>
        </div>
    </li>
</ul>
</div>

</div>

```

The preceding code is used for setting up the slider shown in the layout of the Home page. We are using the **Flexslider** slider for that, and it requires us to markup our HTML like the previous code.

```

</div>

<div class="row-fluid fguid">
    <div class="span4 you-click">
        <h2>You Click</h2>
        
        <ul>
            <li>Upload a photo and write a message</li>
            <li>Fill out the addresses</li>
            <li>Buy one at a time or start an account</li>
        </ul>
    </div>
</div>

```



```
<div class="span4 we-send">"
    <h2>We Send</h2>
    "
    <ul>
        <li>Your custom Foldagram is printed</li>
        <li>We stamp it for you</li>
        <li>Foldagrams are mailed weekdays at noon</li>
    </ul>
</div>
<div class="span4 they-fold">"
    <h2>They Fold</h2>
    "
    <ul>
        <li>Open the Foldagram</li>
        <li>Fold into standing position</li>
        <li>Skip the fridge magnets and picture
            frames!</li>
    </ul>
</div>
</div>
```

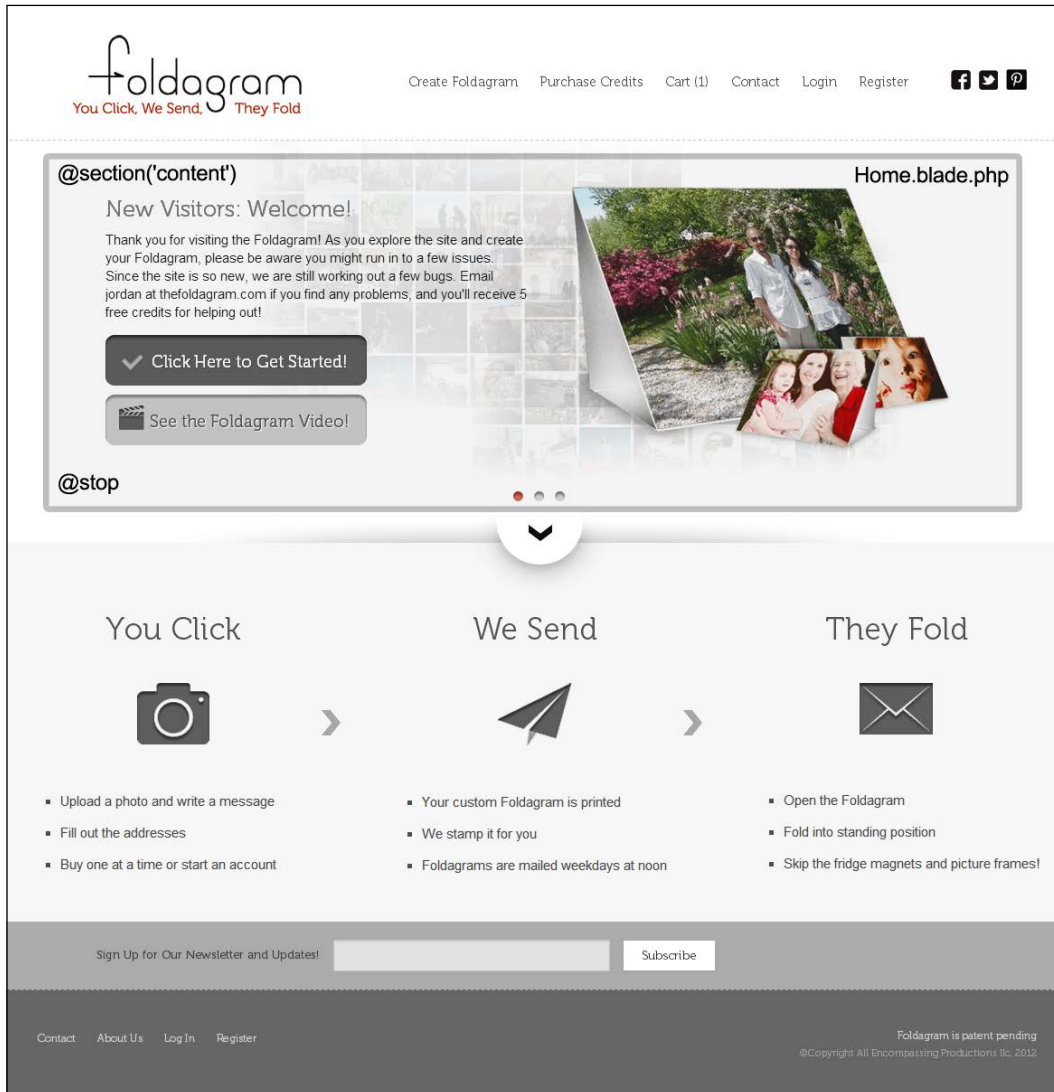
The previous section is used for three sections of the Home page displayed in the layout:

```
@stop
```

As we can see from the source code, we are using `@extends` which will extend the default layout from the layouts directory. We have the section content defined, which will yield an extended default layout. Our Home page content section has a slider code. Now all that is left is adding route to the Home page. So let's do that. The following code in `routes.php` will do that:

```
Route::get('/', function()
{
    return View::make('home')->with('title', 'The Foldagram')-
        >with('page', 'home');
});
```

So, we are using the Laravel's Blade Template engine to work out the layout template for our site. Here is what we have done:



Templating with Laravel

The purpose here is to understand how the Laravel templating works. Also, we will learn how to create a general layout template. Although we are not reviewing every single detail in the code, but our goal is to create the Laravel template out of the code.

You might be wondering what about CSS/JS? For consistent layout, we are using the Twitter bootstrap (<http://twitter.github.io/bootstrap/>). It is a CSS framework that you might already be familiar with. For the sliders part of the code, we are using Flexslider (<http://flexslider.woothemes.com/>). Once you include the Flexslider JS/CSS in your code, all you have to do is create a div container with an unordered list. And each list item can contain your description and image, which we have seen in `app/views/home.blade.php`.

The JavaScript code required to trigger the Flexslider is given as follows. It goes in `public/js/global.js`.

```
$(window).load(function() {  
    $('.flexslider').flexslider({  
        animation: "slide",  
        slideshow: false,  
    });  
});
```

Here is a stylesheet which is required for the Home page styles. It goes in `public/css/style.css`.

```
body { font-size: 14px; color: #777; font-family: 'Museo'', Arial;  
line-height: 20px; }  
p { margin-bottom: 15px }  
.da-slide p { color: black }  
a:hover { color: #e65c08; text-decoration: none; }  
li { line-height: 25px }  
h3 { font-size: 18px; line-height: 20px; }  
h4 { font-size: 16px; font-weight: normal; }  
.btn { font-family: Museo; font-weight: normal; }  
.row { margin: 0 }  
.header { border-bottom: 1px dashed #d1d1d1; padding-bottom: 25px; }  
.header .logo { padding-top: 20px; padding-left: 5%; }  
.header .logo a:hover { border: none }  
.header .menu { padding-top: 80px; width: 54%; }  
.header .menu ul { float: right; margin: 0; }  
.header .menu li { display: inline; padding: 0 10px; }  
.header .menu li a { color: #646363; font-size: 15px; font-family:  
'Museo', "'Myriad Pro', 'Droid Sans'", sans-serif; }  
.header .menu li a:hover { text-decoration: none; color: #646363 ;  
border-bottom: 1px solid #e33f23; }  
a { color: #e33f23 }  
.footer a:hover { color: #848484; border-bottom: 1px solid #e33f23; }  
.header .social { text-align: right; margin-left: 10px; padding-top:  
78px; width: 10%; }  
.header .social a { margin-right: 5px }
```

```

img.facebook { width: 22px; height: 22px; background: url(..img/
facebook.png) 0 0; }
a:hover img.facebook { background: url(..img/facebook.png) 0 -22px }
img.twit { width: 21px; height: 22px; background: url(..img/twit.png)
0 0; }
a:hover img.twit { background: url(..img/twit.png) 0 -22px }
img.ping { width: 24px; height: 23px; background: url(..img/ping.png)
0 0; }
a:hover img.ping { background: url(..img/ping.png) 0 -23px }
.slider { overflow: hidden; background: url(..img/sliderbg.png) no-
repeat 275px 0; }
.slider-content { background: url(..img/middle-shadow.png) no-repeat
center bottom; padding-bottom: 85px; position: relative; z-index: 1; }
.fguid { background-color: #f9f9f9; padding-top: 75px; padding-bottom:
40px; margin-top: -60px; text-align: center; }
.fguid h2 { margin: 0; text-align: center; font-family: 'Museo',
'''Myriad Pro', 'Droid Sans''', sans-serif; font-size: 35px; color:
#6f6f6f; font-weight: normal; margin-bottom: 40px; }
.fguid ul { margin-top: 50px }
.fguid ul li { text-align: left; list-style: square; font-family:
Arial; font-size: 15px; color: #696969; margin-bottom: 10px; text-
shadow: 0px 0px 1px rgba(0,0,0,0.2); }
.fguid ul li:last-child { margin-bottom: 0 }
.fguid .we-send,
.fguid .you-click { background: url(..img/right-arrow.png) no-repeat
right 110px }
.you-click ul,
.we-send ul,
.they-fold ul { margin-left: 60px }
.fguid .they-fold img { margin-bottom: 10px }
.subscribe-form { background-color: #bcbcbc; padding: 20px 0 0; }
.subscribe-form .subscribe-content { padding-left: 100px }
.subscribe-form .subscribe-content label { color: #585858; font-size:
13px; text-shadow: 0px 0px 1px rgba(0,0,0,0.2); margin-right: 15px; }
.subscribe-form .subscribe-content .input-large { background-color:
#e7e7e7; border: 1px solid #d2d2d2; border-radius: 0; height: 25px;
margin-right: 15px; width: 292px; }
.subscribe-form .subscribe-content .btn { background-color: #fff ;
background-image: none; color: #454444; border-radius: 0; height:
33px; padding-left: 20px; padding-right: 20px; border: 0; }
.subscribe-form .subscribe-content .btn:hover { background-color:
#e33f23 ; color: #fff; }
.footer { background-color: #808080; padding-top: 40px; border-top:
1px dashed #bcbcbc; padding-bottom: 38px; }
.footer .footer-menu ul li { display: inline; list-style: none;
padding: 0 10px; }

```

```
.footer .footer-menu ul li a { color: #d5d5d5; text-shadow: 0px 0px
2px #999999; font-size: 12px; }
.footer .footer-menu ul li a: hover { color: #d5d5d5 ; text-
decoration: none; border-bottom: 1px solid #e33f23; }
.footer .copyright { text-align: right; padding-right: 30px; }
.footer .copyright h4 { font-size: 12px; text-shadow: 0px 0px 2px
#999999; color: #d5d5d5; margin-top: 0; margin-bottom: 0px; font-
weight: normal; }
.footer .copyright p { color: #b3b3b3; font-size: 11px; margin-bottom:
0; }
.inner-top { background: url(../img/inner-top.png) no-repeat right
top; background-color: #eeeeee; margin-top: 1px; padding-bottom: 20px;
position: relative; }
.inner-top h2 { color: #777; font-size: 22px; font-family: 'Museo',
""Myriad Condensed Web", 'Droid Sans""', sans-serif; line-height:
24px; font-weight: normal; text-shadow: 0px 0px 2px #fff; margin-top:
0; }
.inner-top p { font-size: 13px; font-family: Museo; color: #848484; }
.inner-top .inner-content { padding-left: 30px; padding-top: 35px; }
.inner-top img { position: absolute; right: 50px; bottom: -65px; }
.home .content .alert { margin-top: 0px; margin-bottom: 75px; }
.content .alert { margin-top: 65px }
.content .dcontent .alert { margin-top: 25px }
.content h2 { font-size: 30px; font-family: 'Museo', ""Myriad
Condensed Web", 'Droid Sans""', sans-serif; font-weight: normal; line-
height: 20px; color: #555555; margin-bottom: 35px; }
.content .dcontent { padding: 0 25px; margin-left: 0; margin-top:
47px; padding-bottom: 100px; }
.header { position: relative; min-height: 111px; }
.header .menu { width: 52% }
```

Setting up the inner pages

For setting up the inner pages, all that we have to do is set up the inner banner section with appropriate markup, so that the pages look as per the design. First, let's set up a route for our **About Us** page:

```
Route::get('/about', function()
{
    return View::make('about')->with('title','About
    Foldagram')->with('page','about');
});
```

Now, as per our route definition, we would need to create `about.blade.php` at `app/views` directory. So, here is our **About Us** page:

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

@section('inner-banner')
<div class="row-fluid inner-top">
  <div class="span6 inner-content">
    <h2>About</h2>
    <p>We have a great story, and we'd love to tell you about
      it. By the way, did you watch the original Foldagram
      video on the front page?</p>
  </div>
  
</div>
@stop

@section('content')
<div class="span12 dcontent">
  <div class="span7 about well">
    <h3>Where it all began...</h3>
    <p> Everyone knows the internet provides a wonderful tool
      for quickly sharing thoughts and photos, but do you ever
      miss real mail? You know, the physical stuff that shows
      up in your actual mailbox, something you can touch.
      Postcards are nice, but they aren't the same as opening
      an envelope. Besides, people seem to toss them out or
      lose them because there's no easy way to stand them up!
    </p>
    <p> This is what led Jordan Bundy to create The Foldagram
      in 2012. He was hiking in the Rocky Mountains near his
      home in Vail, CO, and wanted to share the beauty of the
      landscape. It seemed like a waste to just put photos on
      Facebook, since they would get lost in the constant
      shuffle of updates. The idea of making prints, writing
      messages, stuffing envelopes and going to the post office
      just seemed daunting. And yet, what other option was
      there?</p>
    <p> After several prototypes, The Foldagram was born.
      Thanks to a successful Kickstarter campaign, you can send
      someone a unique piece of physical mail as easily as
      updating a blog.</p>
    <p> If you have a media query, please contact Jordan at
      thefoldagram.com. Otherwise, feel free to browse what's
      already been written!</p>
  </div>
  <div class="span5 well">
```

```
<h3>What 's' been said</h3>
<span>Web Links</span>

    <p><a href="http://offbeathome.com/2012/10/the-
      foldagram">"Offbeat Home</a></p>
<p><a
  href="http://www.coloradocountrylife.com/index.php?
  option=com_content&view=article&id=1068:send-a-
  letter&catid=39:other-resources&Itemid=91">"Colorado
  Country Life</a></p>

  <h4>Audio Clips</h4>

  <p><a
    href="http://www.youtube.com/watch?v=QT0POw0DFr8">"
    The second interview with Ski Country</a></p>
<p><a
  href="http://www.youtube.com/watch?v=MQlObuRkJrk">
  "The first interview with Ski Country</a></p>
</div>
</div>
@endsection
```

Here, we are extending the same default template that we have used. But, we have one extra inner banner section in our About view, which is for the top banner we have in the **About Us** page. Then the content section will contain the About Us page content divided in two columns. This is how we can create any number of inner pages.

Creating the newsletter section

Do you remember we have a newsletter section in our template? It's time to make the newsletter section work. First of all we would need to create a subscribe table and a migration file. So, let's generate a migration file via the artisan command:

```
php artisan migrate:make create_subscribe_table --table=subscribe --
create
```

Also, add an e-mail column into the generated file. So, it will look like the following code:

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

class CreateSubscribersTable extends Migration {
    public function up()
    {
```

```

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

    public function down()
    {
        Schema::drop('subscribers');
    }
}

```

Let's run a migration to generate a table from migration file:

php artisan migrate

Now, as we have the database table ready for it, let's explore how we can save the user's e-mail when users subscribe to our site. We have formed a code in the template for the newsletter section, which is given as follows:

```

<div class="row-fluid subscribe-form">
    <div class="span12 subscribe-content">
        {{ Form::open(array('url' => 'subscribe')); }}
        {{ Form::label('something', 'Sign Up for Our Newsletter
            and Updates!'); }}
        {{ Form::text('email', '', array('class' => 'input-
            large', 'placeholder' => '')); }}
        {{ Form::submit('Subscribe'); }}
        {{ Form::close(); }}
    </div>
</div>

```

Here, `Form::open` will generate the `<form>` tag with the `post` method and the correct `post` URL where the form will be posted. So, we need to have a route which responds to subscribe the `POST` request. So, let's define that:

```

Route::post('/subscribe', function()
{
    $input = Input::all();
    $rules = array('email'=>'required|email');

    $validation = Validator::make($input, $rules);
    if ($validation->passes())
    {
        Subscribe::create($input);
    }
}

```




```
        return Redirect::to('home')->with('success', 'Thanks' for
            signing Up Foldagram.');
```

```
    }

    return Redirect::to('home')
        ->withInput()
        ->withErrors($validation)
        ->with('message', 'There were validation errors.');
```

```
    });
```

Here, our route file will receive the POST request for the subscribe route. First, we will check with the validation object that the e-mail field is not empty and the e-mail format is valid. If validations are passed, then we will store e-mail to the subscribe table via the `Subscribe Eloquent` object. So, let's create an Eloquent model for our subscribe table.

 Eloquent is Laravel's **(ORM Package) Object Relational Mapper Package**. It maps the database tables to the classes and database table columns as the class attributes, which allows us to do very useful things to the model such as saving, editing, and deleting.

Our model will reside at `app\models\Subscribe.php`:

```
class Subscribers extends Eloquent {
    protected $fillable = array('email');
}
```

Here, we are defining a fillable array which is used by Eloquent to determine which fields need to be allowed to change via the `Subscribe` model. Eloquent will make sure when you are using `Subscribe::create($input)`; even if the `$input` array has other fields, it will not be saved in the database. Only the e-mail field will be saved as we have declared it in the fillable array.

Now if validation fails, we are redirecting the user with `Redirect::route('home')`. It will redirect user with input and errors contained in the validation object. We need to print these errors somewhere in our template; so let's add a message container in our default template at `/app/views/layout/default.blade.php`.

```
@if ($errors->any())
    <ul>
        {{ implode('', $errors->all('<li class="error">: "message</li>')) }}
    </ul>
@endif
```

This will print any error messages received via the validation object. So that's all we need to make the Newsletter section work. Users can fill e-mail from anywhere in the site and it will be added to our newsletter table via the code we just wrote.

Creating a Foldagram form

It's time to move on to creating our Foldagram form, but first let's check the designs that we have got.

The image shows a 3D foldagram form overlaid on a website. The form is a grey, box-like structure with various input fields and buttons. The background shows the website's layout with a header, navigation links, and content sections.

Header: foldagram logo, "You Click, We Send, They Fold", navigation links (Contact, Login, Register), and social media icons (Facebook, Twitter, Pinterest).

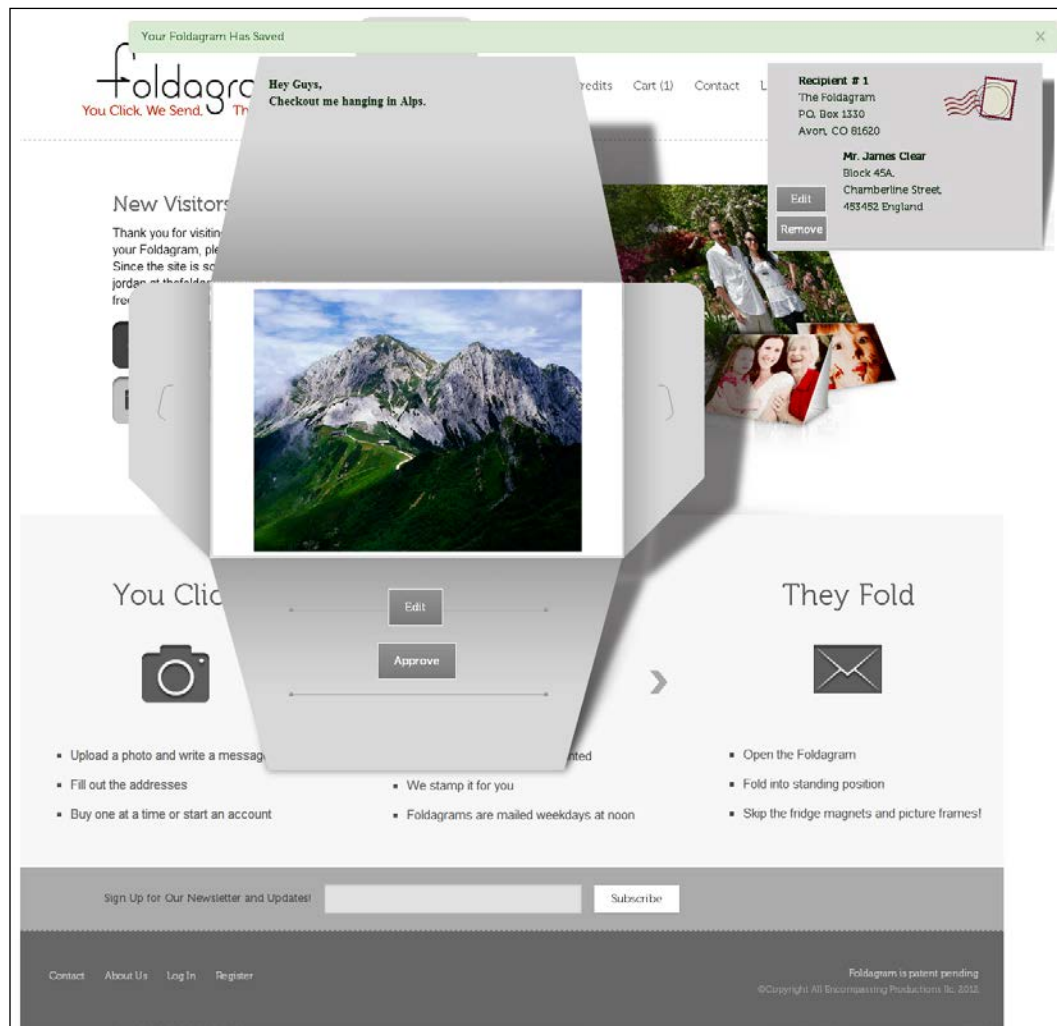
Form Fields:

- Message:** "Hey Guys, Checkout me hanging in Alps." (1162 chars left)
- Upload File:** "Files must be less than 8 MB. Allowed file types: png gif jpg jpeg"
- Recipient's Address - 2:**
 - Full Name*
 - Enter Recipient's Address here *
- Buttons:** "Add Another Address", "Save"

Footer: "Sign Up for Our Newsletter and Updates!" with a text input field and a "Subscribe" button. Navigation links (Contact, About Us, Log In, Register) and copyright information (Foldagram is patent pending, ©Copyright All Encompassing Productions Inc. 2012).

Foldagram form

Now when the user submits the form, we need to display the preview form that is something as follows:



Foldagram preview

If you remember our user story, a user can create Foldagram from anywhere in the site. So anytime a user presses the create Foldagram link in the top navigation, we need to open the model box displayed in the preceding figure. Also, when the user submits that form, we need to present the information from the *Foldagram form* screenshot into the *Foldagram preview* screenshot, which is the main purpose of our site. The *Foldagram preview* screenshot displays exactly how the Foldagram will look and at which address it will be sent.

So first of all, we need to add a modal box. This would need to be in every page as the user can create a Foldagram from any page. In order to do this, we will create a subview and then we will include that subview in our default template. So let's create our subview at `app/views/Foldagram.blade.php`:

```
<div id="popup" class="modal hide fade in" tabindex="-1"
  role="dialog" aria-labelledby="myModalLabel" aria-
  hidden="false" style="display: block;">
  <button type="button" class="close " data-dismiss="modal"
    aria-hidden="true">x</button>
  <div class="modal-body">
    <div class="cfrom-wapper">
      {{ asset('img'/create-form-flow.png') }}
      {{ Form::open( array('url' => URL::route('create')) ) }}
    </div>

    <div class="setp1">
      <div class="message create-form">
        <textarea rows="4" class="enter-message
          required" placeholder="Enter Your Message:
          *" name="message" cols="50"></textarea>
        <br>You have <span id="charsLeft">1200</span>
          chars left.
        <div class="clear"></div>
        <div id="thumbnail">
          {{ asset('img'/placeholder.gif) }}
        </div>
        <input type="file" style="display:none"
          id="upload-image" name="image"
          class="required">
        <div id="upload" class="drop-area">
          <span class="uploadfile">Upload File</span>
          <p class="help-block">Files must be less
            than <span>8 MB</span></p>
          <p class="help-block">Allowed file types:
            <span>png gif jpg jpeg</span></p>
        </div>
      </div>
    </div>

    <div class="address">
      <div class="photocreate-form
        recipient_address_wapper">
        <div class="recipient_address" id="recip_1">
```

```

        <h3>Recipient's Address <span
            class="account" style="display: none;">- "
        1</span> </h3>
        <input placeholder="Full Name* : "
            class="required " type="text"
            name="add[0][fullname]" value=""> "
        <textarea placeholder="Enter Recipient's
            Address here * : " rows="8"
            class="required "
            name="add[0][address_one]"
            cols="50"></textarea>
    </div>
</div>
</div>

<div class="submit">
    <div class="submit-content">
        <button class="add btn-large btn"
            type="button">"Add Another Address</button>
        <button class="remove btn-large btn"
            type="button" style="display:
            none;">"Remove Address</button><br>
        <button class="submit-btn btn-large btn"
            type="submit">"Save</button>
    </div>
</div>

    {{ Form::close() }}
</div>
</div>
</div>
```

And include this subview in our main template by adding the following code at the bottom of our default.blade.php file located at /apps/views/layout.blade.php.

```

</div><!-- End Container -->
@include('foldagram')
```

So, now we have this Foldagram pop up which routes at the \create route. We need to add the JavaScript code to open it as a modal window. In order to do this, I have added a jquery ui modal window script and the following script in global.js:

```

$($('.create-foldaram').click(function() {
    $('#createfoldagram').modal({
        keyboard: false
    });
});
```

With the preceding code, we are hooking the click event with the create Foldagram link in navigation. We are opening a model window and loading the content of our pop up. So basically, we have set up our Foldagram code.

Summary

In this chapter, we learned how we can set up a project in Laravel, how templating system works, and how to utilize Laravel's structure to create a web application.

We started with migrations and created basic migrations for our Foldagram project. We learned how to prepare schema and how we can seed the database.

Next, we learned how to set up layouts and Blade templating to manage dynamic sections of our pages. Later, we created a newsletter and Foldagram form of our Foldagram application.

In the next chapter, we will work on adding the Foldagram to the cart with the option of previewing and then ordering it. We will create cart package first and integrate this package in our Foldagram pop up.

5

Creating a Cart Package for Our Application

In this chapter we will learn how Laravel handles packages. We will see the core concepts of the Laravel framework to understand the relationship between the packages and IoC container. We will also learn how to create packages.

Here is the list of things we will cover in this chapter.

- The IoC container
- Dependency Injection
- Service providers
- Packages in Laravel 4
- Facades
- Creating a Cart package
- Preview Foldagram
- Edit Foldagram
- Delete Foldagram

One of the most appealing aspects of Laravel 4 is packages. It's the central idea of Laravel 4 and hence it is built with lots of packages. All these packages are completely separated from each other. In fact, some of the packages come from the **Symfony** framework. So as a developer, you can develop the Eloquent package or a log package if you wish to. If you think your changes can benefit the community, you can send a pull request to your changes via GitHub. And if it's approved, anybody can download it via the `Composer Update` command.



You can check Laravel packages by searching GitHub for packages such as illuminate/Auth or illuminate/Log. You can fetch the Git repository of these packages to see the actual source, or if you want to work on them, you can work through these packages.

So how does Laravel work? How can it be so decentralized that its individual packages can be updated, and how does it act as a framework? What's the thing that binds all these packages? The simple answer to all these questions is **IoC container**.

Introducing IoC container

IoC means **Inversion of Control**. You can define how a particular class or interface can be resolved in the IoC container. The IoC container will use that information whenever you need to resolve a class or interface throughout your application. Here, the main benefit is that if you ever decide to rewrite the implementation of a class, or you need to change the class with a new class, the IoC container will resolve the class or interface to your implementation as long as your implementation follows the original implementation's interface.

I know what most of you feel right now. It's very difficult to understand what the IoC container, or even certain other terms, mean as we generally don't face these terms in our day-to-day lives as PHP developers. So let's try to break down things to a simpler level. Why do we need the IoC container? That brings us to one more difficult question to be answered in order to achieve Dependency Injection.

Dependency Injection

Dependency Injection is a software-design pattern which is used to remove hard dependencies from code, and it allows us to change that dependency without affecting the actual implementation.

Let's make it simpler with an example. We have our Car class defined as follows:

```
class Car
{
    function __construct()
    {
        $this->_database = DB::getInstance();

        // OR infamous Global $DB way

        Global $db;
```

```

        $this->_database = $db;
    }

}

```

Here, database is our hard dependency. We have bound the `Car` class with a database object. Assume that you want to use the `Car` class in one of your other projects and it uses a different database such as MongoDB or MSSQL. Apart from this, you also have to create a singleton DB object to access your data, even when your database implementation is different. Or that you need to have a global variable `$db` set in such a way that the `Car` class can use it. So how can we remove this dependency? Here is the answer:

```

class Car
{
    private $_database;

    function __construct($dbconnection)
    {
        $this->_database = $dbconnection;
    }
}

```

So if you have observed carefully, we are now injecting our database dependency via constructor. That's what we call Dependency Injection. Now it doesn't matter how the implementation of the database class is used; all we need is a database object passed at our constructor, and it will work on any type of project you develop in future. As you may have the `mongodb` class or the `mysql` class as your database, you can just pass this class via constructor. The `Car` class will use that as a database object. You can instantiate your `Car` class as shown in the following code:

```
$car = new Car($databaseconnection);
```

There is one more way we can set dependencies. Let's develop our `Car` class further in the following manner:

```

class Car
{
    private $_database;
    private $_engine;

    function __construct(){

    }

    public function setDBConnection($dbconnection){

```

```
        $this->_database = $dbconnection;
    }

    public function setEngine($engine){
        $this->_engine = $engine;
    }

}
```

As you can see, we are using setter methods to set our dependencies now. To instantiate the Car class now, we would need the following code:

```
$car = new Car();
$car->setDatabase($databaseConnection);
$car->setEngine($engine);
```

This is how we can manage our dependencies. Now, imagine you have lots of classes created, but as you create classes you will have more dependencies. Each time you want to access the class, you will need to instantiate dependencies; this is a cumbersome procedure. You will find that your code often becomes messy and much harder to read due to dependencies. To solve this problem, you can use containers. Containers are created for managing all the dependencies of a particular class or application.

```
class Container {

    public static $_database;

    public static function BuildCar($engine) {

        $car = new Car();
        $car->setDatabase(self::$_database);
        $car->setEngine($engine);

        return $car;
    }

}
```

As you can see, all our dependencies are loaded when we call the static function BuildCar within the Container class. We don't need to set it exclusively each time we need to use our Car class.

As we are creating all our dependencies within our container, we can simply load our `car` instance wherever we want by using the following line of code:

```
$car = Container::BuildCar();
```

That's how a container can be loaded. Remember our definition; containers are used for a particular class or an interface to resolve instances application wide. Here, we are using it for our `Car` class.

So, we have learned two important aspects of creating a `Cart` package: Dependency Injection and IoC containers. Let's try to learn things in the context of Laravel. How does Dependency Injection work in Laravel? How can you use it in your projects?

The IoC container in Laravel resolves classes or interfaces throughout Laravel. You can look at the code of the IoC container at the root of your project, `/vendor/laravel/framework/src/Illuminate/Container.php`. You can use the `bind` method of container to bind our classes so that we can use it throughout our Laravel application.

Here is how you can resolve a class dependency in the IoC container, provided you are auto-loading your class:

```
App::bind('Car', function()
{
    $car = new Car();
    $car->setDatabase($databaseConnection);

    return $car;
})
```

So, whenever you load your `Car` class, all its dependencies will automatically be loaded by the IoC container. This is one of the most useful things Laravel provides, because in your typical web application, you will use a lot of classes or third-party classes. When you want to write a wrapper around your third-party class, or you are developing your own package, you can ensure that when your wrapper class loads, all its dependencies are loaded automatically via the IoC container.

Now that we know what the IoC container and Dependency Injection are, it's time to know where we can put this code. Well, you can put your IoC container bindings in the two places listed below:

- In your `app/start/global.php`
- Service providers

You can place your container bindings directly in `global.php` file and Laravel will pick up your binding when it bootstraps itself. This approach works for one or two classes but when you have lot of classes or interfaces your `global.php` file will be cluttered with lots of messy container bindings. A better way to handle container bindings is service providers.

Service providers

A service provider class in Laravel registers the IoC container bindings. It's the class that will bind your package classes with the Laravel IoC container. Generally, this can be done using two methods: `register` and `boot`. The `register` method will be called by the Laravel framework of any service provider in the `app.php` config file, at `app/config/`, during the bootstrap part of Laravel. So you can bind your classes with Laravel's container.

Let's look at the sample service provider in the following code snippet:

```
use Illuminate\Support\ServiceProvider;

class CarServiceProvider extends ServiceProvider {

    public function register()
    {
        $this->app['Car'] = $this->app->share(function($app)
        {
            $car = new Car();
            $car->setdb($dbconnection);
            return $car;
        });
    }
}
```

Our `ServiceProvider` has to extend Laravel's `ServiceProvider` class, and for that we will be using the `Illuminate\Support\ServiceProvider` namespace to import the `ServiceProvider` class and extending it via our `CarserviceProvider` class.

Each service provider will have at least a `register` method which allows us to hook our class into the IoC container. In the preceding example, we define that whenever the `Car` class is instantiated within our app, we need to resolve it by creating a new instance of the `Car` class and setting its dependency.

Service providers are the way Laravel registers all its packages. In fact, you can check `app.php` in your `app/config` folder for the `providers` array. If you look at the `providers` array, you will find following lines:

```
'providers' => array(

    'Illuminate\Foundation\Providers\ArtisanServiceProvider',
    'Illuminate\Auth\AuthServiceProvider',
    'Illuminate\Cache\CacheServiceProvider',
    .....
),
```

Remember, at the beginning of the chapter we were wondering how Laravel works its magic? This is how Laravel's magic works:

- Laravel has a lot of core components
- When any request hits the Laravel framework, it bootstraps and loads its `Application` class
- Laravel's `Application` class extends the `container` class, so the `container` class is also loaded
- Framework loads and executes the `register` method of all the service providers that are in the `providers` array at `app/config/app.php`
- Each service provider by the Laravel packages has a registered method that binds each of the package classes with the IoC container
- Once each package is bound with the IoC container, you can use packages such as `Auth`, `Cache`, and `Log` in your application

So, we have learned how the IoC container and service providers help Laravel work together. Now let's move to packages and see if what we have learned helps us build our first package.

Packages in Laravel 4

Packages are the easiest way you can add third-party code in Laravel. Also, you can create your own packages to add new features to Laravel. You can find packages from <https://packagist.org>. There were more than 10,000 packages available, at the time of writing this book, on the website. You will find packages such as the date/time manipulation package, the `carbon` or `requests` package, which will allow you to easily consume APIs in your project.



Most of the packages that you will find at `packagist.org` are standalone packages. These will work with all frameworks. But there are Laravel-specific packages that specially utilize Laravel structures such as Controller, route, Model, and migrations, which will work only with the Laravel framework.

Let's see how we can create a new package via Laravel. To create a new package, we can use the `artisan` command. But we will have to set a few configuration options in `app/config/workbench.php`. In this file you need to set name and e-mail options. If you have set up these values, then fire the following `artisan` command:

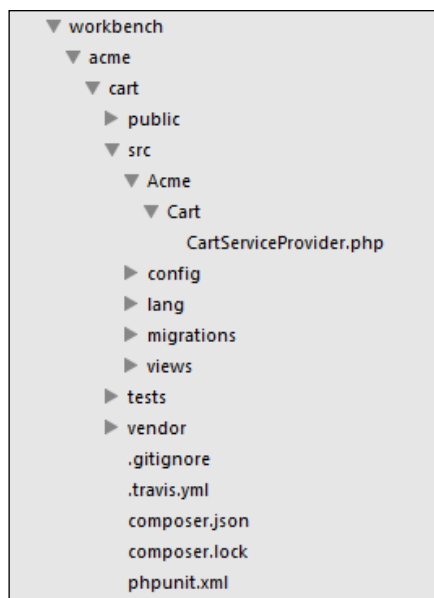
```
$ php artisan workbench Acme/Cart --resources
```

The preceding command will generate a basic skeleton of your Laravel package in the `workbench` directory. If you are wondering what `Acme/Cart` is, `acme` could be your company's name or your name and `cart` could be the package we are trying to create. This is useful to avoid conflicts when more than one author is developing a `Cart` package and one of them has the package name `Acme` as a vendor name.

The `resources` flag tells Laravel to generate a Laravel-specific package. Laravel will add its structure directories, such as `config`, `lang`, `migrations`, and `views`.

Package structure

Here is an image that displays the `workbench` package structure in Laravel 4:



Let's review the package structure. Laravel creates a directory structure based on the vendor (package name) hierarchy. Here we have `Acme/ Cart` as our package name, so Laravel has created the `Acme` directory inside the `Cart` directory. Here is a table displaying the purpose of each file/directory in the package:

File/Directory	Purpose
<code>Public</code>	This manages package assets such as JS, CSS, and images. It is useful because you will need to send your assets while publishing your packages.
<code>Src</code>	This is the source directory of our package. Here, all of our custom classes for packages will be stored. Laravel automatically creates namespace directories so that we can manage our source files in our namespace.
<code>src/ acme/ cart/ cartserviceprovider.php</code>	Laravel generates a service provider file for our package with all the boilerplate code so that we don't have to manually create the service provider code.
<code>Src/Config</code>	This directory is used for creating package-specific configurations.
<code>Src/Migrations</code>	This directory is used for storing all of our package migration. When somebody downloads your package, they can add a package-specific table schema from the migration files in this directory.
<code>Src/views</code>	This directory is used for creating package-specific views.
<code>Tests</code>	This directory will hold all the unit tests you write for your package.
<code>Vendor</code>	This directory will store all the dependent packages that will be required to run our package.
<code>Composer.json</code>	This file will have information specific to your package, such as the package description and package dependencies.
<code>Phpunit.xml</code>	This file is used to run unit tests for our package via artisan.

So now that we know the structure of our package, let's move further to create a basic test case, called *hello world*, for our package. When someone invokes our package as shown:

```
Cart::message();
```

It should return the infamous `Hello World` message from the `Cart` package.

To do that we would need to first create our Cart class. So let's add our Cart class to `workbench/acme/cart/src/Acme/Cart/Cart.php`. The following code shows the creation of the Cart class:

```
namespace Acme\Cart;

class Cart
{
    function message()
    {
        echo "hello world from cart package";
    }
}
```

If you see, we are not using the static method in the preceding code; then how can we call it via `Cart::message()`? Well, we have created our code, but as we have learned, we will have to tie our code with the Laravel IoC container via service provider. Let's create a `CartServiceProvider.php` file, in our Cart package at `src/Acme/Cart` in the following manner:

```
namespace Acme\Cart;

use Illuminate\Support\ServiceProvider;

class CartServiceProvider extends ServiceProvider {

    /**
     * Indicates if loading of the provider is deferred.
     *
     * @var bool
     */
    protected $defer = false;

    /**
     * Bootstrap the application events.
     *
     * @return void
     */
    public function boot()
    {
        $this->package('acme/cart');
    }

    /**
     * Register the service provider.
     */
}
```

```
*
* @return void
*/
public function register()
{
    $this->app['cart'] = $this->app->share(function($app)
    {
        return new Cart;
    });
}

/**
 * Get the services provided by the provider.
 *
 * @return array
 */
public function provides()
{
    return array('cart');
}

}
```

The `register` method of our service provider will bind our `Cart` class with the IoC container, and whenever we access the `Cart` class, it will resolve the `register` method automatically. But you may be wondering about the two other methods: `boot` and `provides`. Well, the `boot` method is used by Laravel to bootstrap the package and load the views, configurations, and other resources. The `provides` method is used to allow services of package to be used via the handle, provided in the arrays. Generally, facades use the handle provided by this method to access the services of package. But we haven't learned what facades is. So let's take a look at it.

Facades

A facade is a class that tapes into the Laravel IoC container and provides access to a class object.

For each Facade class, we need to write a method `getFacadeAccessor()`, as the Facade class of Laravel implements `\Illuminate\Support\Facades\Facade`.

Facades are the reason the Laravel syntax is super clean and yet works; a facade provides static interfaces to classes that are available in the IoC container. So, when you are writing the following code, you are actually calling the View alias, which resolves by the View facade located at `Illuminate\Support\Facades\View.php`:

```
return View::make('hello');

namespace Illuminate\Support\Facades;

class View extends Facade {

    /**
     * Get the registered name of the component.
     *
     * @return string
     */
    protected static function getFacadeAccessor() { return 'view'; }

}
```

As you can see, we have to implement `getFacadeAccessor`, which returns our view object from the IoC container. This allows us to use our View package as a static interface, and we can call methods of our `Illuminate\View\View.php` class easily. Whenever we do that, the IoC container loads dependencies automatically so our syntax is clean and elegant.

Suppose you had to write a code to access the `make` method. We would have to instantiate our dependencies first. The View class in Laravel depends on five classes: `EngineResolver`, `PhpEngine`, `BladeCompiler`, `FileViewFinder`, and `Environment`. Your code would have looked something like the following:

```
$view = new Illuminate\View\View( Environment $environment,
EngineInterface $engine, $view, $path, $data = array() );

$view->make('hello');
```

Imagine if you had to write the preceding code each time you want to load! But you don't have to because of Laravel's magic; that's the power of the Laravel IoC container, service provider, and facades. They all may seem to be complicated at the first instance, but together they form a system that will help you maintain your code in a way that you can write individual packages and yet it will have easy-to-understand and elegant syntax.

To create a facade for your package, you will need to extend the `Facade` class provided by the `Laravel` class. Now, let's create our `Cart` facade so that we can access it just like the `Laravel` packages. This is done in the following manner:

```
namespace Acme\Cart\Facades;

use Illuminate\Support\Facades\Facade;

class Cart extends Facade {

    /**
     * Get the registered name of the component.
     *
     * @return string
     */
    protected static function getFacadeAccessor() { return 'cart'; }

}
```

Now we need to add our alias to the `alias` array in `app/conf/app.php` so that we don't have to write down the full namespace and can easily access our `Cart` class. This is done in the following manner:

```
'aliases' => array(
    'Cart'          => 'Acme\Cart\Facades\Cart'
);
```

Now that we have set up everything, let's write a route to test if it works in our `routes.php` file:

```
Route::get('/cart', function()
{
    return Cart::message();
});
```

If everything works out well, you will be greeted by `hello world` from `cart` package.

We have learned how we can create a package and benefit from it. Now let's use our newly gained knowledge to create a `Cart` class for our `Foldagram` web application.

Cart functions

Let's plan out what we will be doing, so when we code, it will be easy for us.

Our Cart class will handle the following options:

- Adding Foldagram to the cart
- Updating the cart
- Deleting from the cart
- Viewing the cart contents
- Viewing the cart total
- Deleting all the items from the cart

Here, Foldagram is our main product. If you remember, in the last chapter we had created a form that will ask for information about Foldagram.

For ease of use, we will create a Cart facade, which will allow us to use our Cart package by simply calling our add method via the Cart alias.

```
$item = array(
    'id'      => 'foldagram_id',
    'qty'     => 2,
    'price'   => 25,
    'name'    => 'Foldagram'
);

Cart::insert($item);
```

To handle the cart transactions temporarily, we will use Laravel's session class. Session will hold the cart contents, as long as it is not ordered.

The Cart class

Let's redefine our Cart class in Cart package to handle the session and initialize necessary parts.

```
namespace Acme\Cart;
use Illuminate\Session\Store as SessionStore;
class Cart
```

```
{

    private $session = null;

    private $container = null;

    private $cart_contents = null;

    public function __construct(SessionStore $session)
    {

        $this->session = $session;

        $this->container = 'Foldagram_Cart';

        $this->initializecart();

    }

}
```

Here we are injecting the `session` class via its namespace to make sure we are not setting a hard dependency on the `session` class. If somebody wants to replace the `session` class with, say for example `MongoDB`, they can easily do that as they will not have to change all of our classes. They can just inject their object, and our class will use that implementation to serve `Cart` via the IoC container.

We are going to use the `container` variable for the purpose of wrapping our cart items. Here is our `initializecart` method for initializing our cart with default values:

```
public function initializecart()
{
    array_set($this->cart_contents, $this->container, $this->session-
    >get($this->container));

    if (is_null(array_get($this->cart_contents, $this->container,
    null)))
    {
        array_set($this->cart_contents, $this->container, array('cart_
        total' => 0, 'total_items' => 0));
    }

}
```

Here, we are setting our `cart_contents` array from session. The `array_set` helper helps us to wrap our cart options array with our container. Here, `array_get` and `array_set` are array helpers provided by Laravel. If you are wondering what they do, here is a very simple example of what they do:

```
$array = array('gallery' => array('title' => 'My image'));
$value = array_get($array, 'gallery.title'); // will output "My Image"

array_set($array, 'gallery.title', 'My Image changed');
$value = array_get($array, 'gallery.title');//will output "My Image
changed"
```

As shown, `array_get()` and `array_set()` can help you access deeply nested arrays easily via the key-value pair chain.

Adding Foldagram to the cart

Now that it's all set up, let's add a method to our `Cart` class for adding Foldagram to it:

```
function insert($item = array())
{
    $this->validate($item);
        $item['qty'] = (float) $item['qty'];
        $item['price'] = (float) $item['price'];

    if (isset($item['id']))
    {
        $rowid = md5($item['id']);
    }

    $item['rowid'] = $rowid;
    $item['qty'] += (int) array_get($this->cart_contents, $this->container . '.' . $rowid . '.qty', 0);

    array_set($this->cart_contents, $this->container . '.' . $rowid, $item);

    $this->update_cart();

    return $rowid;
}
```

Here we are setting initial items, such as quantity, price, and row ID, from array. Row ID is generated by the `md5` method for creating a unique row for our cart contents, so later we can use this to update our record or delete it from our container. After updating individual items, we are merging the `$item` array into our container.

The first line has the `validate` method for validating items that go into the container. Here is the code for that:

```
function validate($item){

    if ( ! is_array($item) or count($item) == 0)
    {
        throw new Acme\Cart\CartInvalidDataException;
    }

    $required_indexes = array('id', 'qty', 'price', 'name');

    foreach ($required_indexes as $index)
    {
        if ( ! isset($item[ $index ]))
        {
            throw new Acme\Cart\CartRequiredIndexException
('Required index [' . $index . '] is missing.');
```

```
        }
    }

    if ( ! is_numeric($item['qty']) or $item['qty'] == 0)
    {
        throw new Acme\Cart\CartInvalidItemQuantityException;
    }
```

```
    if ( ! is_numeric($item['price']))
    {
        throw new Acme\Cart\CartInvalidItemPriceException;
    }
```

```
}
```


In our `validate` method, we first check if the array is empty; if it's empty, we will throw an exception. Next, we check if all our indexes are present in our item array. And lastly, we check `qty` and `price` for the numeric values.

Now, if you have noticed, we have the `update_cart` method in our `insert` method. It's the method that will update our cart total and put our data into the session. Let's define it in the following manner:

```
function update_cart()
{
    array_set($this->cart_contents, $this->container . '.total_items',
0);
    array_set($this->cart_contents, $this->container . '.cart_total',
0);

    foreach (array_get($this->cart_contents, $this->container) as $rowid
=> $item)
    {
        if ( ! is_array($item) or ! isset($item['price']) or !
isset($item['qty']))
        {
            continue;
        }

        $this->cart_contents[ $this->container ]['cart_total'] +=
($item['price'] * $item['qty']);
        $this->cart_contents[ $this->container ]['total_items'] +=
$item['qty'];

        $subtotal = (array_get($this->cart_contents, $this->container .
'.' . $rowid . '.price') * array_get($this->cart_contents, $this-
>container . '.' . $rowid . '.qty'));

        array_set($this->cart_contents, $this->container . '.' . $rowid .
'.subtotal', $subtotal);
    }

    $this->session->put($this->container, array_get($this->cart_
contents, $this->container));
    return true;
}
```

In the preceding code, we will initially merge our total items and cart total with our container. Then we will loop through each item and update our cart total and total items in the cart. And finally we will use the `session` library's `put` method to put our data into the session, so our data is preserved between requests.

So the exact process we are using is as follows:

1. First we create a container that holds the cart via the class constructor.
2. We inject the session via the constructor so it can be utilized at any place in the class.
3. Next, we check if session has the `cart` array, and if it does, we wrap it with our container via `array_get`.
4. For adding the item to our cart, we use the `item` array. For each item, we add a unique row so we can manage when we want to edit or delete items.
5. Then, we update the cart totals and the number of the cart items so that we can display them immediately. And finally, we save it in a session.

As we have gone through the process of adding Foldagram to our cart, let's move to the part where we update it.

Updating the cart

If someone changes our Foldagram quantity, we need to update our cart to make sure our price is updated. We also need to track if somebody updates the quantity as 0; this means we need to remove that Foldagram from our cart. So here is our updated method for working on that:

```
function update($item = array())
{
    $rowid = array_get($item, 'rowid', null);

    if (is_null($rowid))
    {
        throw new CartInvalidItemRowIdException;
    }

    if (is_null(array_get($this->cart_contents, $this->container . '.' .
        $rowid, null)))
    {
        throw new CartItemNotFoundException;
    }
}
```

```
    }

    $qty = (float) array_get($item, 'qty');

    array_forget($item, 'rowid');
    array_forget($item, 'qty');

    if ( ! is_numeric($qty) )
    {
        throw new CartInvalidItemQuantityException;
    }

    if ( ! empty($item) )
    {
        foreach ( $item as $key => $val )
        {
            array_set($this->cart_contents, $this->container . '.' . $rowid
                . '.' . $key, $val);
        }
    }

    if (array_get($this->cart_contents, $this->container . '.' . $rowid
        . '.qty') == $qty)
    {
        return true;
    }

    if ($qty <= 0)
    {
        array_forget($this->cart_contents, $this->container . '.' .
            $rowid);
    }

    else
    {
        array_set($this->cart_contents, $this->container . '.' . $rowid .
            '.qty', $qty);
    }

    $this->update_cart();

    return true;
}
```

Here, we first fetch `rowid` from the array so that we can get a handle for our `item` array. If it isn't there, a request could be forged, as we are creating that parameter via our `insert` method. Next, we check via `array_get` if `rowid` is in our container array. If it's not, that means the cart item is not in our cart, so we must generate an exception with `CartItemNotFoundException`.

Now we check if our newly provided `item` array is empty or not. If it's not empty, we can safely update each of our `Cart` class items from the `item` array via `array_set`. Then we check if our quantity is 0; if it is, we need to remove it from our `Cart` class. If everything is in the right order, our `update_cart()` method will be called and will hold our `Cart` class container array into the session; this is how we can update our cart items.

Deleting from cart

To delete an item from cart, all we need is the `rowid` of that item. Once we have the `rowid` (remember, we have already written the code in our `update` method), if the cart item has 0 as a quantity, it should be removed from the cart. So here is the code for deleting a cart item:

```
public function remove($rowid = null)
{
    if (is_null($rowid))
    {
        throw new CartInvalidItemRowIdException;
    }

    if ($this->update(array('rowid' => $rowid, 'qty' => 0)))
    {
        return true;
    }
}
```

As you can see in the preceding code, if `rowid` is passed, we just need to call our `update` method with the `item` array as `rowid` and the quantity as 0. Our `update` method will handle this and remove it from our cart, as well as update the session as it already has a mechanism for that.

Viewing the cart contents

We have created methods for adding, editing, and deleting items from cart. Now, let's create a method that will allow us to get the cart data as an array. This is done in the following manner:

```
public function contents()
{
    $cart = array_get($this->cart_contents, $this->container);

    array_forget($cart, 'total_items');
    array_forget($cart, 'cart_total');

    return $cart;
}
```

In order to display our cart array, we first fetch it from our container via the `array_get` Laravel helper function. Next, we remove items such as `total_items` and `cart_total` just so that we can easily loop over our cart array.

Viewing the cart total

Next, we need to get the total of our cart, which includes each item's price and quantity. However, we are already managing it in our cart container array as we are updating the total when we insert or update any item from the cart. So, to load the cart total, all we have to do is execute the following code:

```
public function total()
{
    return array_get($this->cart_contents, $this->container .
        '.cart_total', 0);
}
```

Deleting all items from the cart

Up next, how do we remove all items from cart or how can we destroy our cart? First, we override our container array with empty values and then remove it from our current session, as shown in the following code snippet:

```
public function destroy()
{
    array_set($this->cart_contents, $this->container, array('cart_total'
        => 0, 'total_items' => 0));

    $this->session->forget($this->container);
}
```

Here, we set our cart with an empty container and then, via Laravel's `session` class' `forget()` method, we remove it from our actual session.

We may need one more method to identify how many items are in cart for displaying it in our navigation. So here is the method that will display the count of cart items in our pages:

```
public function total_items()
{
    return array_get($this->cart_contents, $this->container .
        '.total_items', 0);
}
```

As we already have the `total_items` key, which updates the total items in the cart when an item is added, edited, or deleted; we don't need to count it, we can grab it from our cart array.

So, this is the basic functionality we needed for our Cart package to work. But we need to rewrite our service provider, as we are passing a dependency of our session. We need to pass it via service provider's `register` method. So here is our new service provider for the Cart class:

```
namespace Acme\Cart;

use Illuminate\Support\ServiceProvider;

class Cart extends ServiceProvider {
    protected $defer = false;

    public function boot()
    {
        $this->package('acme/cart');
    }

    public function register()
    {
        $this->app['Cart'] = $this->app->share(function($app) {
            return new Cart($this->app['session']);
        });
    }

    public function provides()
```

```
{  
    return array('Cart');  
}  
  
}
```

If you've noticed, in the `register` method, we resolve our `session` array by injecting it into the `Cart` class via `$this->app['session']`. So Laravel will make sure that whenever a new instance of `Cart` class is created, an instance of the `session` class will be injected into our `Cart` class. We have already created facades of our `Cart` class and a reference of it in `app/config/app.php`, so it will be automatically loaded by Laravel whenever the page requests the `Cart` class.

Integrating the Cart package in Foldagram order process

In *Chapter 4, Building a Real-life Application with Laravel – The Foldagram*, we have created `Foldagram Lightbox`, which allows a user to create `Foldagram`. Now, it's time to integrate features, such as adding `Foldagram` to database and preview `Foldagram`, as well as to add into the cart to manage orders.

- Adding the `Foldagram` information to the `Foldagram` table
- Adding the recipient information to the recipient's table
- Adding the `Foldagram` order details to the cart
- Editing the `Foldagram` information
- Editing the recipient information
- Deleting `Foldagram` from the cart
- Creating the preview page to preview the order

Adding the Foldagram information to the Foldagram table

So let's start by adding the `Foldagram` information to the `Foldagram` table. We need to set up two things here: first, we need to save the message to the `Foldagram` table; then, we need to process the `Foldagram` image provided by the user and resize it so that we can use it when we want to preview it. But we would also need to set up two tables: `Foldagram` and `Foldagram_Recipients`.

Here is the migration command for adding a Foldagram schema in our database migrations:

```
$ php artisan migrate:make create_foldagram_table
```

This will generate a schema file at `app/database/migrations`. Here is the schema we need to add to our Foldagram table schema:

```
Schema::create('foldagram', function(Blueprint $table)
{
    $table->increments('id');
    $table->text("message");
    $table->string("image",255);
    $table->boolean('status', array('0', '1'))->default(0);
    $table->integer('user_id');
    $table->timestamps();

});
```

Similarly, here is schema for the recipient's table:

```
Schema::create('foldagram_reff_address', function(Blueprint $table)
{
    $table->increments('id');
    $table->integer('foldaram_id');
    $table->string("fullname",255);
    $table->string("country",255);
    $table->string("address_one",255);
    $table->string("address_two",255);
    $table->string("city",255);
    $table->string("state",255);
    $table->string("zipcode",255);
    $table->timestamps();

});
```

Next, we need to migrate the tables into the database via the following command:

```
$ php artian migrate
```

To access these tables, we need to create models. So let's create two models at `app/models/Foldagram.php` and `app/models/Receipients.php`:

```
class Foldagram extends Eloquent {

    public function recipients()
```



```
{
    return $this->hasmany('recipients');
}
```

The following is our recipient's Model, which is similar to our Foldagram Model:

```
class Recipients extends Eloquent{

    public static $table = 'recipients';

    public function foldagram()
    {
        return $this->belongsTo('Foldagram');
    }

}
```

If you noticed, we have defined an Eloquent Model relationship between the two tables. We have defined our Foldagram Model to have a one-to-many relationship with the recipient's Model. This is going to help us when we want to query related data from both the tables. Now, we have set up the basic requirements and can query the database or add/update/delete the database via our models.

Let's add a post route to submit Foldagram Lightbox to the Foldagram's create Controller:

```
Route::post('create', 'foldagram@create');
```

Now, let's add the code to save the message and generate `Foldagram_id` in the following manner:

```
function create(){

    $foldagram = new Foldagram;

    $foldagram->message = Input::get('message');

    $foldagram->save();

}
```

Image resizing in Laravel

We also need to save the Foldagram image and resize it for previewing it in thumbnails. For resizing the images, we will use an external package called **Intervention/image**. To use the intervention package, we need to install it via Composer. Open your `composer.json` file and add the following line:

```
"intervention/image": "dev-master"
```

And run following Composer command:

```
$ Composer install
```

This will install our package in the vendor directory. Also, we need to add the service provider of the class so we can bind it to our Laravel setup in `app/config/app.php` in the service provider array.

```
'Intervention\Image\ImageServiceProvider'
```

Also we need to add the facade, so we can easily access it via an alias in our alias array:

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

Now, we will have access to our intervention library via the image alias autoloaded by Laravel. To store an image, we will create the `uploads` directory in our `public/img` directory. Also, for storing thumbnails, we will create a `thumbnails` directory in our `public/img` directory. Here is the code, which we will add code to our `create` method of the Foldagram Controller, to save the image first and then resize it.

```
$filename = $foldagram->id."_".str_random(8). '.' . File::extension(Input::file('image.name'));

$destinationPath = 'img/uploads/';
$thumbnailPath = 'img/thumbnails/'.$filename;

Input::file('image')->move($destinationPath, $fileName);

Image::make(Input::file('image')->getRealPath())->resize(100,
100)->save($thumbnailPath);

$foldagram->image = $filename;
$foldagram->save();
```

In the preceding code, we first create a unique image path so that the image upload doesn't conflict even if another image has the same name. Our Model is `$foldagram` and `$foldagram->id` is our newly inserted ID. We then move the image via the `Input::move` method. We use our `Image` package to resize the image to 100 x 100. In the end, we use our `Foldagram Eloquent Model` to save the generated filename to our database.

Adding the recipient information to the Recipients table

Now, let's add the recipient information to the `Recipients` table. A user can add as many recipients as they want, so we need to loop over all the recipient text inputs and save them to the database. Let's add the following code to our `create` method:

```
$recipients_input = Input::get('address');

if(!empty($recipients_input)){

    $recipients = array();

    foreach($recipients_input as $value){

        $recipients[] = array(
            'fullname'=>$value['fullname'],
            'address_one'=>$value['address_one'],
        );

    }

    $foldagram->recipients()->save($recipients);
}
```

Here, we loop through all the inputs named `address` and then add it to our `recipients` table via our `Foldagram Model`. Laravel will automatically add a `Foldagram ID`, as we are using the `Foldagram Model` and have the relationship defined in our models.

Adding the Foldagram order details to our Cart package

Now it's time to add our order details into our Cart package. We need to gather all our inputs and create an array so that we can pass it into our `Cart::insert($array)` method. Let's add the following code to our Foldagram Controller's `create` method:

```
try{

    $qty = count(Input::get('add'));

    $item = array(
        'id'      => $foldagram->id,
        'qty'     => $qty,
        'price'   => Config::get('foldagram.price'),
        'name'    => 'Foldagram'
    );

    // Add the item to the shopping cart.
    Cart::insert($item);

    // Redirect to the cart page.
    return Redirect::route('home')
        ->with('success', "Your Foldagram Has Saved")
        ->with('redirect', "preview");

} catch(Exception $e) {

    return Redirect::route('home')->with('error', $e->getMessage() );
}

catch (Cart\CartInvalidDataException $e)
{
    return Redirect::route('home')->with('error', 'Invalid data
passed. ');
}

catch (Cart\CartInvalidItemQuantityException $e)
{
```

```
        return Redirect::route('home')->with('error', 'Invalid item
quantity.');
```

```
    }

    catch (Cart\CartInvalidItemRowIdException $e)
    {
        return Redirect::route('home')->with('error', 'Invalid item row
id.');
```

```
    }

    catch (Cart\CartInvalidItemNameException $e)
    {
        return Redirect::route('home')->with('error', 'Invalid item name.');
```

```
    }

    catch (Cart\CartInvalidItemPriceException $e)
    {
        return Redirect::route('home')->with('error', 'Invalid item
price.');
```

```
    }
}
```

If you notice, we are taking the Foldagram price from the config file `app/config/Foldagram.php`, which will have the following code:

```
Return array('price' => 5)
```

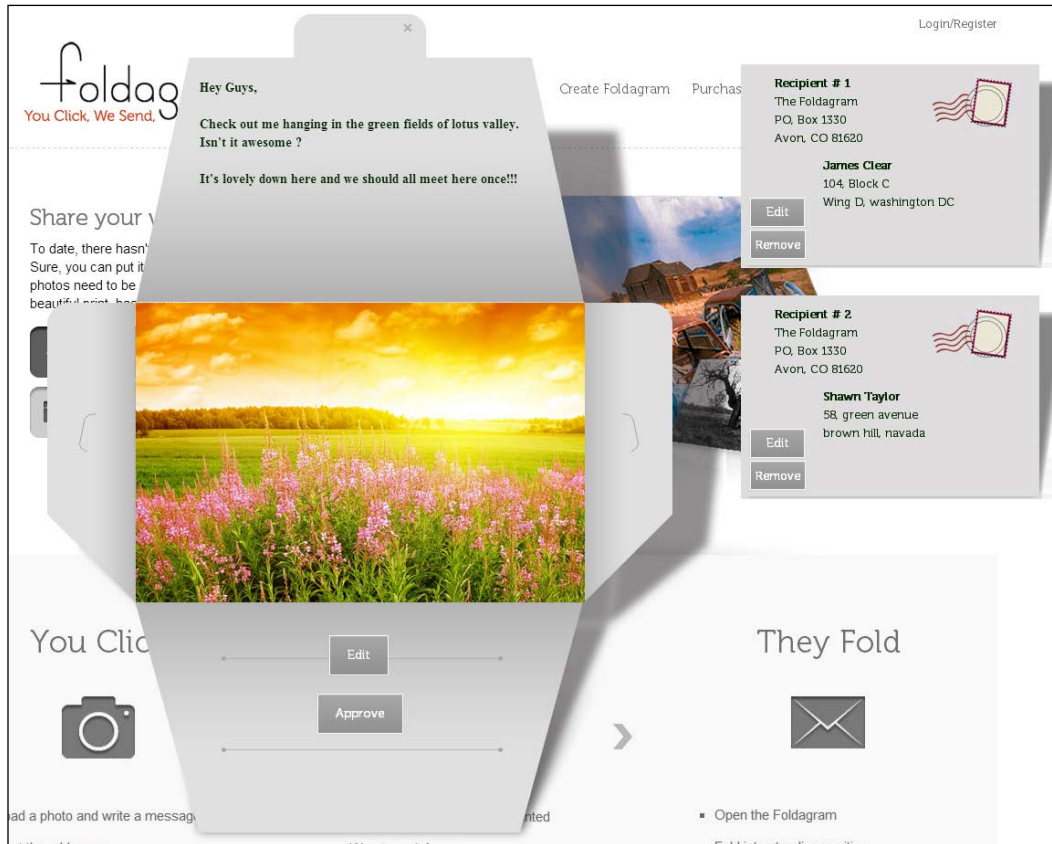
Here, USD 5 is the price per Foldagram. We load it with Laravel's `Config::get` method. Next, we build our array for adding it to our Cart package. As we have already loaded the Cart package via service provider and given an alias via facade, we can directly use the `Cart::insert` method to temporarily store our cart into session via the Cart package. Isn't it sweet?

If you noticed, there are lots of exceptions. These exceptions come from our Cart class. We were throwing these exceptions in the `insert` method of our Cart class section. And we are catching these package exceptions here and displaying the appropriate errors.

If everything goes right, we will send the user to the home page with a preview parameter, which will trigger the preview process, as users can get instant feedback on how things will look. At the end of the chapter, we will learn in detail about how the preview will work. Right now, let's see how we can edit our Foldagram or edit the recipient information.

Creating the preview page to preview Foldagram

As soon as we add Foldagram, we are redirecting users to our preview Lightbox in the add Foldagram section. Here is the design of our preview Foldagram section:



Now, as per the preview design page, we need to build a Lightbox pop up. It will hold the preview information and the recipient information that we fetch from the database. As you can see, we need to give the user control to edit the Foldagram recipient addresses, the message, and the picture information in the design. So, let's first construct our Foldagram preview Lightbox.

We need to add our code into our `app\views\layouts.blade.php`, as we want to allow the preview page Lightbox pop up to open on any page of our site. We will use our jQuery modal component that we used for creating the Foldagram Lightbox. But for the purpose of readability, we will create our view file at `app\views\foldagram_preview.blade.php` and just include this file by using the following code:

```
@include('foldagram_preview');
```

Here is our JavaScript code which will display the modal window. The modal function is called via the jQuery UI library:

```
$(document).ready(function() {  
    $('#preview').modal('show');  
});
```

And finally, to preview the image, here is the Lightbox code that will generate the Model container with data, which will be stored in the `Foldagram_preview.blade.php` file.

```
<?php  
$foldagram_data = Foldagram::find($id);  
?>  
  
<div class="preview">  
    <div class="cfrom-wapper">  
        <div class="vertical">  
             }}">  
            <div class="textmessage ">  
                {{ $foldagram_data->message }}  
            </div>  
            <div class="imgprivev">  
                {{ HTML::image('img/uploads/' . $foldagram_data->path, "Foldagram  
Image") }}  
            </div>  
            <div class="submit">  
                <div class="submit-content">  
                    <button class="edit-btn btn-large btn-large btn"  
type="button">Edit</button>  
                    <button class="approve-btn btn-large btn-large btn"  
type="button">Approve</button>  
                </div>  
            </div>  
        </div>  
    </div>  
</div>
```

In the preceding code, we first find Foldagram via our Eloquent Model Foldagram. Then, we displayed it in our preview container, which is opened via the jQuery UI Modal box. We show the message, as well as uploaded image, via our Eloquent objects `$foldagram_data->message` and `$foldagram_data->path`.

Now, we also need to preview the recipient addresses attached with Foldagram. To do so, we need to find the addresses from the database. Eloquent is here to help; we can find these addresses with just one line of Eloquent, which is as follows:

```
$Receipients = Foldagram::find(1)->Receipients;
```

Now, we need to loop them and display them in our model box. Here is the code for that:

```
<div id="raddress" class="scroll-pane-os jspScrollable">
  @foreach($Receipients as $value)
  <div class="address_list" id="{{ $value->id }}">
    <div class="fromaddress">
      <p><strong>Recipient # {{ $i++ }}</strong></p>
      The Foldagram<br/>
      PO, Box 1330<br/>
      Avon, CO 81620
    </div>
    <div class="toaddress">
      <div class="dispaddress">
        <p> <strong>{{ $value->fullname }}</strong><br/>
        <?php $text = preg_replace("/[\r\n]+/", "\n", $value-
>address);
        $text = wordwrap($text,120, '<br/>', true);
        echo $text = nl2br($text); ?>
      </p>
    </div>
  </div>
  <a href="{{ URL::to('remove/'. $value->id.'/'. $rowid) }}"
class="removeadd btn btn-danger">Remove</a>
  </div>
  @endforeach
</div>
```

In the preceding code, we loop through each recipient and display their address as a box in our Model window. We extract the recipient's name and address via `$value->fullname` and `$value->address`. We process the address value, as we want it to be more than 120 characters, otherwise it will fall off from the box; we are converting line breaks from `textarea` to HTML line breaks via the `nl2br` function.

Deleting the recipient's information

In the Foldagram preview, we provide the option to delete recipients from Foldagram. In our Foldagram Controller let's add the `removeReceipient` method for that with the following code:

```
function removeRecipient($id)
{
    $Receipient = Receipient->find($id);
    $Receipient->delete();
    return Redirect::route('preview');
}
```

Now, we just need to add a route for this in our `routes.php` file.

```
Route::get('remove/{:any}', array('as' => 'removereceipient', 'uses'
=> 'foldagram@removeRecipient'));
```

Editing the Foldagram information

First, let's add a route for edit in our `routes.php` file:

```
Route::post('edit', 'foldagram@edit');
```

Let's add code to save the edited information in our Foldagram Controller:

```
public function edit()
{
    $foldagram_id = Input::get('foldagram_id');
    $foldagram = Foldagram::find($foldagram_id);

    if(Auth::check()){
        $user_id = Auth::user()->id;
    }
    else {
        $user_id = 0;
    }

    $foldagram->message = Input::get('message');
    $foldagram->user_id = $user_id;
    $foldagram->save();
}
```

Here, we will get the Foldagram ID from the request, and we are using it to fetch the data from the Foldagram model's `find` method. We also check if the user is logged in. If the user is logged in, we save his ID into the Foldagram table so we can always display this item in the user's cart. We also save the message, just in case it's edited by the user via the Foldagram Model's `save` method.

Deleting Foldagram from the cart

We need to allow the user to remove Foldagram from cart. Let's add a code for that in our Foldagram Controller:

```
function remove($item_id = null, $id=null)
{
    try
    {
        if(!empty($id)){
            Cart::remove($item_id);
        }else {
            return Redirect::to('cart')->with('error', 'Invalid foldagram
ID!');
        }
    }

    catch (Cart\CartItemInvalidRowIdException $e)
    {
        return Redirect::to('cart')->with('error', 'Invalid Item Row
ID!');
    }

    catch (Cart\CartItemNotFoundException $e)
    {
        return Redirect::to('cart')->with('error', 'Item was not found in
your shopping cart!');
    }

    catch (Cart\CartException $e)
    {
        return Redirect::to('cart')->with('error', 'An unexpected error
occurred!');
    }

    return Redirect::to('cart')->with('success', 'The item was removed
from the shopping cart.');
```

Here, we first check if the ID is present; if it is, we remove the cart item from our Cart package via the `remove` method. We check for various exceptions, such as, whether the item is in the shopping cart or if the item's row ID is present. If everything goes right, we send the user a success message on our cart page.

Summary

In this chapter we have learned how Laravel packages work and how to make one that will work for you. We also learned how to integrate our standalone package in our complex Foldagram project and do things the Laravel way.

In the process, we learned about Dependency Injection and how IoC containers can manage dependencies for you. We also learned how service providers integrate your package with IoC containers and we used all these concepts to build our Cart package.

In the next chapter we will learn about the user's dashboard.

6

User Management and Payment Gateway Integration

In this chapter, we will create our user registration, login, and dashboard pages. We will learn how we can authenticate users. Later, we will learn integrating our cart with payment gateway. By doing this, we will also learn how we can utilize the existing Laravel packages, as well as how to find them and integrate them in our projects.

So after completing cart pages, I have to meet Jordan to show him the progress and planning the user section of the web application. After getting a lot of exciting demo time, Jordan was told now it's time to move to the user section of the site.

After our user section's discussion, we decided that the following list is essential to the user section:

- User should be able to register
- User should be able to log in
- User dashboard should be provided where a user can see all his account options
- User should be able to change the password
- User should be able to manage his profile

We worked through and finalized the user section wireframes so the designer can give us the design on time and I can utilize those designs when I create pages. We also discussed how the workflow of cart pages and checkout page works as well as our payment gateway. Jordan was thinking of using PayPal but we decided finally on Stripe as we both agreed it's the simplest and best way to integrate credit cards on our application.

So now we need to create the user section with the following pages:

- User registration
- User login
- User dashboard
- User password change
- Manage profile

We will create a layout for the user section as all the user features would have same look and feeling. We will store all the user section views in a separate user folder and create a special layout file for the section; so in future if we need to apply changes to the section, we can easily apply it via user layout.

Introducing the Sentry package

But how do we authenticate users? Well Laravel 4 does come with built-in authentication mechanism, but it doesn't include the ability to handle user groups and permissions. So, do we build the authentication system?

The answer is No! Because as we have discussed previously, Laravel is a *psr-0* ready framework and it can utilize thousands of packages available at <https://packagist.org/>. One of the consistently managed authentication library for Laravel 4 is Sentry 2. It's very flexible and provides a great way to authenticate the users.

So let's quickly install Sentry 2 into our project. The first step to install Sentry is to add its entry to the `composer.json` file. In our `composer.json` file's `require` attribute, we will add the following code:

```
"require": {  
    "laravel/framework": "4.0.*",  
    "cartalyst/sentry": "2.0.*"  
}
```

Run `composer update` to install the Sentry package into our project. Next we need to add `SentryServiceProvider` into our application's service providers' list so that Laravel can tap into the Sentry object whenever we need authentication. So add the following code into `app/config/app.php` in the service provider array:

```
Cartalyst\Sentry\SentryServiceProvider
```

The last step is optional but allows us to write Sentry without specifying namespace, so we will add the Sentry alias into our `app/config/app.php` as follows:

```
'Sentry' => 'Cartalyst\Sentry\Facades\Laravel\Sentry'
```

Also don't forget to install the schema from Sentry. To add all the database schemas from Sentry, we will run migrations via the following command:

```
php artisan migrate --package=cartalyst/sentry
```



You should always use the artisan command, `php artisan config:publish vendor/package`, for creating a copy of the configuration files of the package in your config directory. As and when you update your package, your vendor directory changes will be lost.

With Sentry, authenticating the users is a breeze! All you have to do is use Sentry's `authenticate` method to check if the user's credentials are true. Here is a sample code which will do that:

```
$credentials = array(
    'email' => 'test@example.com',
    'password' => 'test',
);
// Try to authenticate the user
$user = Sentry::authenticate($credentials, false);
if($user){ echo "You are logged in!"; }
```

As we have loaded Sentry via its service provider, we can access the Sentry package methods directly. To authenticate the user, we need to provide an array with an e-mail and a password.

Setting up our user section

To set up our user section, we will create user layout and the following files in the views directory:

- `/view/layouts/user.blade.php`: Main user layout file
- `/view/user/register.blade.php`: Registration page
- `/view/user/login.blade.php`: Login page
- `/view/user/changePassword.blade.php`: Change password page
- `/view/user/dashboard.blade.php`: Dashboard page

Here all the pages will extend the `user.blade.php` file. Let's create our `user.blade.php` file.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
  <title>{{ $title }}</title>
  <meta name="viewport" content="width=device-width">
  <link href="//fonts.googleapis.com/css?family=Droid+Sans:400,700"
rel="stylesheet" type="text/css">

  {{ HTML::style('css/bootstrap.css') }}
  {{ HTML::style('css/style.css') }}
  {{ HTML::style('css/user.css') }}

</head>
<body class = "{{ $class }}">

  <div class="container">
    <div class="row-fluid header">
      @if(Sentry::check())
        <p class="userlink" > {{ link_to_route('myaccount', 'My
Account') }}</p>| Welcome !
      @endif
    </div>
    @yield('inner-banner')
    <div class="row-fluid content">
      @yield('content')
    </div>
    <div class="row-fluid footer">
      <div class="span8 footer-menu">
        <ul>
          <li>{{ link_to_route('contact', 'Contact') }}</li>
          <li>{{ link_to_route('about', 'About Us') }}</li>
          <li>{{ link_to_route('login', 'Log In') }}</a></li>
          <li>{{ link_to_route('register', 'Register') }}</a></li>
        </ul>
      </div>
      <div class="span4 copyright">
```

```

        <h4>Foldagram is patent pending</h4>
        <p>&copy;Copyright All Encompassing Productions llc, 2012</
p>
        </div>
    </div>
</div><!-- End Container -->

</body>
</html>

```

Here we are including the `user.css` file for the user section and in the header we are including a link to my account only if the user is logged in. We are checking it via the `Sentry::check()` method; it will ensure that the user is logged in so the link will be only displayed to the logged in users. Then we are including a banner and the main containers which can be filled by our custom pages.

Register user

Now as we have a skeleton for the user section ready, let's create our registration screen. We would need to create a route and controller method for that. Here is our route method for registration:

```

Route::get('register', array('as' => 'postregister', 'uses' => 'pages@
register'));
Route::get('myaccount', array('as' => 'myaccount', 'uses' => 'pages@
myaccount'));

```

Now we need to create the `register` method that will load the registration view. Here is code for that:

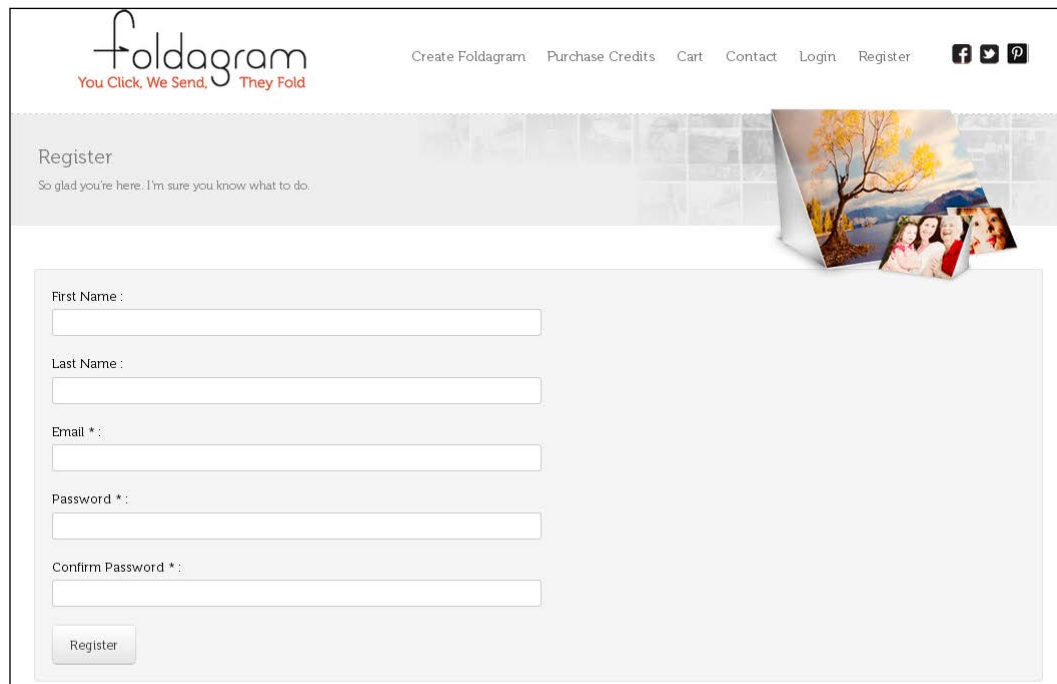
```

function register()
{
    if(Sentry::check()){
        return Redirect::to('myaccount');
    }
    return View::make("user.register")->with("title","The Foldagram -
Register")
        ->with("page_title","Register")->with('class','register');
}

```

We are checking if the user is already logged in; if yes, then instead of showing the user registration page, we are redirecting the user to my account page. If the user is not logged in, then we are loading our registration blade page file from the `user` directory in the `views` folder.

To construct the view file, we would need to know its layout and structure. Here is the layout of our registration page:



The screenshot shows the registration page for 'Foldagram'. The header includes the logo and navigation links: 'Create Foldagram', 'Purchase Credits', 'Cart', 'Contact', 'Login', and 'Register'. There are also social media icons for Facebook, Twitter, and Pinterest. The main content area is titled 'Register' and has a sub-header 'So glad you're here. I'm sure you know what to do.' Below this is a registration form with the following fields: 'First Name:', 'Last Name:', 'Email *:', 'Password *:', and 'Confirm Password *:'. Each field has a corresponding input box. At the bottom of the form is a 'Register' button. To the right of the form is a banner image showing a tree and a group of people.

User registration layout

Based on our registration layout, let's code `view/user/registration.blade.php`. First let's add the inner-banner section as follows:

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

@section('inner-banner')
    <div class="row-fluid inner-top">
        <div class="span6 inner-content">
            <h2>{{ $page_title }}</h2>
            <p>So glad you're here. I'm sure you know what to do.</p>
        </div>
        
    </div>
@stop
```

Here we are extending our user layout and displaying the banner section. We are printing the title of the page that we send using the Controller.

Now let's add our main registration form section via the content section:

```
@section('content')
<div class="span12 dcontent">
  {{ Form::open( array('url'=>'register') ) }}
  <div class="control-group">
    <label class="control-label" for="first_name">First Name :</
label>
    <div class="controls">
      <input type="text" name="first_name" class="input-xxlarge"
id="first_name" value="{{ Input::old('first_name') }}" placeholder="">
    </div>
  </div>
  <div class="control-group">
    <label class="control-label" for="last_name">Last Name :</
label>
    <div class="controls">
      <input type="text" name="last_name" class="input-xxlarge"
id="last_name" value="{{ Input::old('last_name') }}" placeholder="">
    </div>
  </div>
  <div class="control-group">
    <label class="control-label" for="rto">Email * :</label>
    <div class="controls">
      <input type="text" name="email" class="input-xxlarge"
id="email" value="{{ Input::old('email') }}" placeholder="">
    </div>
  </div>
  <div class="control-group">
    <label class="control-label" for="password">Password * :</
label>
    <div class="controls">
      <input type="password" name="password" class="input-xxlarge"
id="password" value="{{ Input::old('password') }}" placeholder="">
    </div>
  </div>
  <div class="control-group">
    <label class="control-label" for="password">Confirm Password *
:</label>
    <div class="controls">
```

```
        <input type="password" name="password_confirmation"
class="input-xxlarge" id="password_confirmation" value="{{
Input::old('password_confirmation') }}" placeholder="">

        </div>
    </div>

    <button type="submit" class="btn btn-large">Register</button>

    {{ Form::close() }}
</div>
@stop
```

Here we have created a form via a Form helper which will be posted at the **Register** link. The `div.control` group containers are Twitter bootstrap classes to visually display the form just like the layout in the *User registration layout* screenshot. We are also using the `Input::old` method to retrieve the field values if in case the form is posted back via validation errors.

Now the form will be posted to the `post_register` method of the pages Controller via route, as shown in the following line of code:

```
Route::post('register', array('as' => 'postregister', 'uses' =>
'pages@register'));
```

We need to ensure the following points when the user is registered:

- Validate the user data
- Save the user into our users table via Sentry
- Send the welcome e-mail to the user

So let's first validate the user data, as follows:

```
$rules = array(
    'email' => 'required|email|unique:users',
    'password' => 'required|confirmed',
    'password_confirmation' => 'required'
);

$input = Input::get();
$validation = Validator::make($input, $rules);

if ($validation->fails()) {
    return Redirect::to('register')->with_input()->with_
errors($validation);
}
```

We are using Laravel's `Validation` class to validate the user data. The `$rules` array will ensure the following: e-mail is not empty, it's a valid e-mail format, and it's already not in our database. The validator will return `true` or `false` on its `fails` method. If it fails, we are redirecting the user with the `Redirect` response with errors and input.

Next is to save the user into our database with `Sentry`. Here is code for that:

```
$user = Sentry::createUser(array(
    'email'      => Input::get('email'),
    'password'   => Input::get('password'),
    'metadata'   => array(
        'first_name' => Input::get('first_name'),
        'last_name'  => Input::get('last_name'),
    )
));
```

`Sentry` has a `createUser` method that allows us to send the array which will be saved into our database. `Sentry` will take care of the `password` field automatically. It will generate a hash for us.

Now we need to send the welcome e-mail to the user. We will use the Laravel's built-in `Mail` class to send e-mail. Here is the code for that:

```
Mail::send('emails.welcome', $data, function($message)
{
    $message->to(Input::get('email'))->subject('Welcome to the
Foldagram!');
});
```

Here we are using the `Mail` class to send an e-mail to the user. The first argument `emails.welcome` is the e-mail template file stored at `views/emails/welcome.blade.php`. Here is code for that:

```
<p>We have received your information and your account has been set
up.</p>

<p>You can log in at "{{ link_to_route('userlogin', 'Login')
}}" " Ready to get started? Purchase credits at "{{ link_to_
route('pcredit', 'Purchase Credits') }}" </p>

<p>If you have any questions, please contact us at info@thefoldagram.
com. We're happy to have you!</p>
```

User login

As we have set up user registration, it's time now to set up user login. We need to allow the user to log in via our login page. So here's the route to register our login page:

```
Route::controller('pages');
```

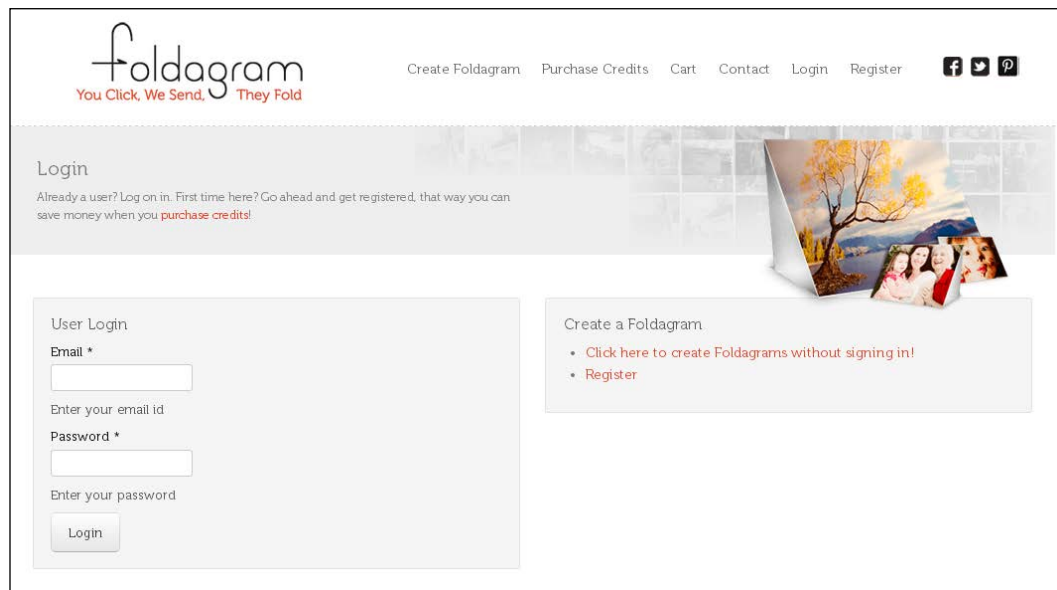
Next is to write a Controller method that will display our login page. Here is the code for that:

```
public function getLogin()
{
    if(Sentry::check()) {
        return Redirect::to('myaccount');
    }

    return View::make("pages.login")->with("title","The Foldagram -
Login")
->with("page_title","Login")->with('class','login');
}
```

Here once again we are checking if the user is logged in; if yes, then instead of displaying the login page, we are redirecting the user to my accounts page.

Here is our login.blade.php layout:



The screenshot displays the 'User Login Layout' for the 'Foldagram' website. The header includes the 'foldagram' logo with the tagline 'You Click. We Send. They Fold', and navigation links: 'Create Foldagram', 'Purchase Credits', 'Cart', 'Contact', 'Login', and 'Register'. Social media icons for Facebook, Twitter, and Pinterest are also present. The main content area is titled 'Login' and includes a sub-header: 'Already a user? Log on in. First time here? Go ahead and get registered, that way you can save money when you **purchase credits!**'. Below this is a 'User Login' form with 'Email *' and 'Password *' input fields, each with a placeholder text 'Enter your email id' and 'Enter your password' respectively, and a 'Login' button. To the right of the form is a 'Create a Foldagram' section with two links: 'Click here to create Foldagrams without signing in!' and 'Register'. The background of the page features a collage of various photos and a 3D photo book.

User Login Layout

Here's the code for our `login.blade.php` page:

```
@extends("layouts.user")

@section('content')
<div class="content-container">
{{ Form::open( array('url'=>'login')) }}
<div class="container">
    <div class="content">
        <div class="box login">
            <fieldset class="boxBody">
                <label>Email</label>
                <input type="text" name="username" tabindex="1" value="{{
Input::old('username')}}" placeholder="">
                <label>Password</label>
                <input type="password" name="password" tabindex="2"
placeholder="">
            </fieldset>
            <div class="footer">
                <input type="submit" class="btn btn-large btn-inverse"
value="Login" tabindex="3">
            </div>
        </div>
    </div>
</div>
@stop
```

Here we are extending our user layout via the `extends` blade method. Also creating the form that will be posted at the **Login** link. Let's create a method which will receive this form and will allow the users to log in to the site.

In our pages Controller, let's add the `post_login` method that will first validate and then allow the user to log in to site.

```
function postLogin()
{
    $rules = array(
        'email' => 'required|email',
        'password' => 'required',
    );
    $input = Input::get();
    $validation = Validator::make($input, $rules);

    if ($validation->fails())
    {
        return Redirect::to_route('userlogin')->with_errors($validation-
>errors)->with_input();
    }
}
```

Here we are checking if the e-mail and the password is supplied and also whether the e-mail format is correct using Laravel's inbuilt `Validator` class. If it's not correct, we are sending the user back to the login page.

Let's check if the user's credentials are right using the following code:

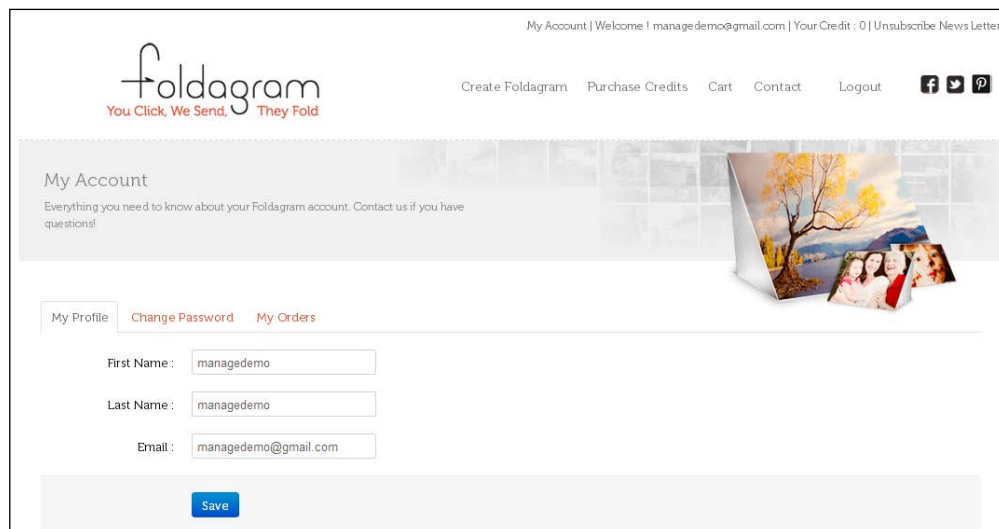
```
$credentials = array( 'email'=> Input::get('email'), 'password'=>
Input::get('password') );

if (Sentry::authenticate($credentials, false))
{
    return Redirect::to('myaccount')
}
else
{
    return Redirect::to('login')->with("error", "There is problem with
login please try again");
}
```

Here we are checking if the user is authenticated using Sentry's `authenticate` method and if the user is authenticated, we are sending the user to my account page.

The User dashboard

The next page we need to create is the user's dashboard page. User's dashboard page layout contains tab control which will hold all three user options. Here is the layout of user's dashboard page.

The screenshot shows a web application interface for 'Foldagram'. At the top, there is a navigation bar with the logo 'Foldagram' and the tagline 'You Click. We Send. They Fold'. To the right of the logo are links for 'Create Foldagram', 'Purchase Credits', 'Cart', 'Contact', and 'Logout', along with social media icons for Facebook, Twitter, and Pinterest. Below the navigation bar, there is a section titled 'My Account' with a sub-header 'Everything you need to know about your Foldagram account. Contact us if you have questions!'. To the right of this text is a large image of a folded card showing a landscape with trees and a sunset. Below the 'My Account' section, there is a tab control with three tabs: 'My Profile' (selected), 'Change Password', and 'My Orders'. Under the 'My Profile' tab, there are three input fields: 'First Name' (containing 'managedemo'), 'Last Name' (containing 'managedemo'), and 'Email' (containing 'managedemo@gmail.com'). Below these fields is a blue 'Save' button.

User dashboard layout

We need to create a tab Controller with three pages for creating the user's dashboard. We would need to register a route for our dashboard page. So here is the code for the dashboard page with the user profile option:

```
Route::get('myaccount', array('as' => 'myaccount', 'uses' => 'pages@myaccount'));
```

Now let's write a Controller method for opening a myaccount view:

```
public function getMyaccount()
{
    if(!Sentry::check()){
        return Redirect::to('login')->with("error", "Please login to
        access your account");
    }

    $user = Sentry::getUser();

    return View::make("pages.myaccount")->with("title","The Foldagram -
    My Account")
    ->with("page_title","My Account")->with('class','myaccount')
    ->with('user',$user);
}
```

We are making sure that the user is already logged in when he is accessing the myaccount Controller method and then we are grabbing the current logged in user's object so that we can display the user information in the view page.

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

@section('content')
<div class="span12 dcontent">
    <div class="tabbable">
        <ul class="nav nav-tabs">
            <li class="active"><a href="#tab1" data-toggle="tab">My
            Profile</a></li>
            <li><a href="#tab2" data-toggle="tab">Change Password</a></li>
            <li><a href="#tab3" data-toggle="tab">My Orders</a></li>
        </ul>
```



```
<div class="tab-content">
  <div class="tab-pane active" id="tab1">
    {{ Form::open( array( 'url' => 'myaccount/profile' ) ); }}
  <div class="control-group">
    <label class="control-label" for="first_name">First Name :</
label>
    <div class="controls">
      <input type="text" name="first_name" id="first_name"
value="{{ $user['metadata']['first_name'] }}" placeholder="">
    </div>
  </div>
  <div class="control-group">
    <label class="control-label" for="last_name">Last Name :</
label>
    <div class="controls">
      <input type="text" name="last_name" id="last_name" value="{{
$user['metadata']['last_name'] }}" placeholder="">
    </div>
  </div>
  <div class="control-group">
    <label class="control-label" for="rto">Email :</label>
    <div class="controls">
      <input type="text" name="email" id="email" value="{{
$user['email'] }}" placeholder="">
    </div>
  </div>

  <div class="form-actions">
    <button type="submit" class="btn btn-primary">Save</button>
  </div>
  {{ Form::close() }}
</div>
</div>
@stop
```

Here we are first having the tab Controller that contains the **My Profile** tab in the opened state. And we are using the user object we sent via the Controller to fill in the values of various fields. Now let's write a method to update the information in the **My Profile** tab.

```
function postProfile(){

    $user = Sentry::getUser();
    $rules = array(
        'email' => 'required|email|unique:users,email, '.$user['id'],
    );

    $input = Input::get();
    $validation = Validator::make($input, $rules);

    if ($validation->fails()) {
        return Redirect::to('myaccount')->with_input()->with_
errors($validation);
    }

    $user_data = array(
        'email'      => Input::get('email'),
        'metadata' => array(
            'first_name' => Input::get('first_name'),
            'last_name'  => Input::get('last_name'),
        ),
    );

    if ($user->update($user_data))
    {
        return Redirect::to('myaccount')->with('success', 'Your
information has been updated successfully.');
```

```
    }
    else
    {
        return Redirect::to('myaccount')->with('error', 'Something went
wrong!..');
```

```
    }
}
```

Here we first get the user's Eloquent object with Sentry's `getUser` method. And then we are validating that the user has updated the e-mail in the correct format using the `Validator` class. Then we are creating the `user_data` array, which holds all the updated fields and with user's `update` method we are updating that information.

Change password

To change the password, we are going to follow the same process as we did in the last two sections. Let's add the code in our `myaccount.blade.php` file in the `tab2` container.

```
<div class="tab-pane" id="tab2">
    {{ Form::open( array('url' => 'myaccount/changepassword') ); }}

    <div class="control-group">
        <label class="control-label" for="old_password">Old Password
    :</label>
        <div class="controls">
            <input type="password" name="old_password" id="old_password"
value="" placeholder="">
        </div>
    </div>
    <div class="control-group">
        <label class="control-label" for="password">New Password :</
label>
        <div class="controls">
            <input type="password" name="password" id="password"
value="{{ Input::old('password') }}" placeholder="">
        </div>
    </div>
    <div class="control-group">
        <label class="control-label" for="password">Confirm New
Password :</label>
        <div class="controls">
            <input type="password" name="password_confirmation"
id="password_confirmation" value="{{ Input::old('password_
confirmation') }}" placeholder="">
        </div>
    </div>
    <div class="form-actions">
        <button type="submit" class="btn btn-primary">Save</button>
    </div>
    {{ Form::close() }}
</div>
```

The preceding code will display three fields, `old_password`, `new_password`, and `confirm new password`. To update the password, let's write a Controller method:

```
function post_changepassword()
{
    $rules = array(
        'old_password' => 'required',
        'password' => 'required|different:old_password|confirmed',
        'password_confirmation' => 'required',
    );

    $input = Input::get();
    $validation = Validator::make($input, $rules);

    if ($validation->fails()) {
        return Redirect::to('myaccount')->with_input()->with_
errors($validation);
    }

    try
    {
        $user = Sentry::getUser();
        if ($user->change_password(Input::get('password'),
Input::get('old_password')))
        {
            return Redirect::to('myaccount')->with('success', 'You
password has been updated successfully.');
```

We are validating the new password and confirm new password is equal as well as old password is provided. If validation passes, we are getting the current user by Sentry's `getUser` method. Then we are using the `change_password` method to change the user's password. This method will automatically check if the old password is correct and if it is correct, it will replace the old password with a new password.

Checkout & payment gateway integration

Jordan reviewed the progress and we both agreed that the user pages look cool. Now we need to focus on the checkout process as well as the payment gateway integration. At that time, Jordan told me he needs to have a credit system. By means of a credit system, he told me he wants to give credit to some of the early beta users and credits will work in the following way:

- Credit is like a point. One point equals one Foldagram.
- Credits will have pricing based on range.
- He can decide credit pricing.
- He can give free credits via backend to some of early users or users who frequently use the site.
- Checkout page needs to have a way to pay via credit.
- Users can buy credits separately via the purchase credit page.


He wants to reward the user who is going to buy Foldagrams in large quantities that is, credit system would work as if you are buying less than five credits, it will cost you 3.75 dollars. But if you are buying more than five credits, it will cost you 3.45 dollars. Higher the credit, lower the price for each credit. It makes perfect sense to him as he had customers who were asking for something like that. So after this sprint, I decided to work out credits workflow in designs of the checkout page with both the designer and Jordan. And we finally decided that our process will work like the user will have the following two options to pay:


- **Pay via credit card:** For this method to work, we decided to use the Stripe payment gateway that helps processing credit cards easily and has a decent API. In the future, we would need to expand on our options.
- **Pay via credits:** Credits can be purchased in bulk and advanced. User can then later send a Foldagram to multiple recipients or multiple Foldagrams to many recipients using this method without having to pay for it every single time.

The designer helped by creating a perfect mockup, which will handle both credit card and credits method. Basically, he provided two radio buttons and based on the radio button clicked, another window will pop up which will ask for the information such as credit card information or will confirm the user that he has enough credits to send Foldagram via credits. Here is the mockup of the page:

Cart

Complete your order, and your Foldagram will be sent soon!



Name	Qty.	Price	Total
Foldagram 	1	4.00	4.00

[Empty the Cart](#)[Continue Shopping](#)

Billing information

Full Name *

Country *

Address 1 *

Address 2

City *

State *

Zip code *

Discount Coupon

Code

Enter discount coupon code.

Payment Option

☐ Use Credit

☒ Credit Card

Payment

Card Owner *

Enter credit card woner.

Card Number *

Enter credit card number.

Expiration *

Enter credit card number.

Security code *

Enter security code.

[Submit](#)

Checkout page layout

Building the checkout page for credit cards

Let's first build our checkout page that will be used for payment via credit card. We will divide our page in three different sections as follows:

- Cart contents
- Order information
- Credit card information

First let's create a Controller which will invoke the checkout page view. Here is the code for that:

```
function getCart() {  
  
    $cart_contents = Cart::contents();  
  
    return View::make('pages.cart')->with('cart_contents', $cart_  
    contents)  
    ->with("page_title", "Cart")->with("title", "The Foldagram - Cart")-  
    >with('class', 'cart');  
}
```

Now as per the Controller method, let's create the cart page in the pages directory under the view directory:

```
@section('content')  
<div class="well">  
<table class="table table-hover table-striped table-bordered">  
    <thead>  
        <tr>  
            <th width="40%">Name</th>  
            <th width="8%">Qty.</th>  
            <th width="12%">Price</th>  
            <th width="12%">Total</th>  
        </tr>  
    </thead>  
    <tbody>  
        @forelse ($cart_contents as $item)  
            <tr>  
                <td> {{ $foldagram_id = '' }}  
  
                <strong>{{ $item['name'] }}</a></strong>  
                <span class="pull-right">
```

```

        <a href="{ { URL::to('cart/remove/' .
$item['rowid']. '/' . $foldagram_id) } }" rel="tooltip" title="Remove the
product" class="btn btn-mini btn-danger"><i class="icon icon-white
icon-remove"></i></a>

    </span>

</td>
<td>
    { { $item['qty'] } }
</td>
<td>{ { format_number($item['price']) } }</td>
<td>{ { format_number($item['subtotal']) } }</td>
</tr>
@empty
<tr>
    <td colspan="6">Your shopping cart is empty.</td>
</tr>
@endforelse
</tbody>
</table>

```

Here we are looping through cart items. We have the `cart_contents` array from the Controller that we are using to loop through the cart items. And we are then using `$item` to display each field from cart.

Next, we need to display order information in our checkout page. Here is the code that we will add to our cart's view file we created in the preceding section:

```

<div class="well">
    <h3> Billing information </h3>
    <label for="fullname"> Full Name * </label>
    <input class="required input-xxlarge" type="text" name="fullname"
id="fullname">

    <label for="country"> Country * </label>
    <input class="required input-xxlarge" type="text" name="country"
id="country">

    <label for="address_one">Address 1 *</label>
    <input class="required input-xxlarge" type="text" name="address_one"
id="address_one">

    <label for="address_two"> Address 2 </label>

```



```
<input class="input-xxlarge" type="text" name="address_two"
id="address_two">

<label for="city"> City * </label>
<input class="required input-xxlarge" type="text" name="city"
id="city">

<label for="state"> State * </label>
<input class="required input-xxlarge" type="text" name="state"
id="state">

<label for="zipcode"> Zip code * </label>
<input class="required input-xxlarge" type="text" name="zipcode"
id="zipcode">

<input type="hidden" name="action" value="foldagram_checkout">
</div>
```

Billing information is formatted with Twitter bootstrap classes to match up to the designer's lookup. Next is the credit card's markup for the checkout page. Here is the code for that:

```
<div class="well credit_card">
  <h3>Payment</h3>
  <div class="payment-error alert alert-error"
style="display:none">
    <button type="button" class="close" data-
dismiss="alert">&times;</button>
    <div class="payment-errors"></div>
  </div>
  {{ Form::label('credit_owner', 'Card Owner *') }}
  {{ Form::xlarge_text('credit_owner', Input::old('credit_owner'),
array('class'=>'required')) }}
  {{ Form::block_help('Enter credit card woner.') }}

  {{ Form::label('credit_card', 'Card Number *') }}
  {{ Form::xlarge_text('credit_number', Input::old('credit_number'),
array('class'=>'card-number required')) }}
  {{ Form::block_help('Enter credit card number.') }}

  {{ Form::label('expiration', 'Expiration *') }}

  {{ Form::select('month', Config::get('application.month') ,Input:
::old('month'),array("class"=>"span1 required card-expiry-month")) }}
  {{ Form::select('year', Config::get('application.year'),Input::old
('year'),array("class"=>"span1 required card-expiry-year")) }}
```

```

        {{ Form::block_help('Enter credit card number.')}}

        {{ Form::label('code', 'Security code *') }}
        {{ Form::span1_text('code', Input::old('code'),
        array('class'=>'card-cvc required')) }}
        {{ Form::block_help('Enter security code.')}}
    </div>

```

Now we have the checkout page's markup generated. To process the information, we would need to save the information into our database and verify the credit card payment and add it once the order is confirmed. So how do we process payments?

Integrating Stripe payment gateway

For processing the Stripe payment gateway, we will use `abodeo/Stripe` package from `packagist.org`. The package is meant for Laravel 4 and allows us to access the Stripe payment gateway API. So let's first install the package.

In `composer.json` add the following line in dependencies:

```
"abodeo/laravel-stripe": "dev-master"
```

And run the Composer update and also add `abodeo` as service provider;

```
'Abodeo\LaravelStripe\LaravelStripeServiceProvider'
```

Then publish the configuration files of package via the artisan command:

```
php artisan config:publish abodeo/laravel-stripe
```

Now we will have a configuration file for the package at `app/config/packages/abodeo/laravel-stripe/stripe.php`. Let's edit that file with the information we get after registration.

```

return array(
    'api_key' => 'my-api-key',
    'publishable_key' => 'my-pub-key'
);

```

Here we need to change the `api` and `pub` keys to the keys given in the account settings by Stripe. Once we update that, we can call the stripe API from our server to process credit cards.

Creating the checkout order process

Now let's create the order process that will charge the user's credit card and save the order into database. Let's create the `checkout` method in our pages Controller for order process.

```
function checkout() {

    $order = new Order;
    $cart_contents = Cart::contents();

    foreach ( $cart_contents as $item) {
        $order->qty = $item['qty'];
        $order->price = $item['price'];
    }

    $order->foldaram_id = Input::get('foldagram_id');
    $order->email = Input::get('email');
    $order->fullname = Input::get('fullname');
    $order->country = Input::get('country');
    $order->address_one = Input::get('address_one');
    $order->address_two = Input::get('address_two');
    $order->city = Input::get('city');
    $order->state = Input::get('state');
    $order->zipcode = Input::get('zipcode');

    $total_amount = Cart::total();
```

Here first we are saving the order into the database via the `order` Model. We are using our `Cart` package to get the cart contents and at last we are calculating the cart total. As we have the cart total, let's write some code to charge the user's credit card for that total.

```
$response = Stripe_Charge::create(
    array(
        "amount" => $total_amount,
        "currency" => "usd",
        "card" => Input::get('stripeToken') ,
        "description" => "Foldagram Payment"
    )
);
```

The preceding code will charge customer's credit card, the total amount that we get from our `Cart` package. Isn't it awesome? Just a few lines of code and our `Cart` package works great together and we can use it to great effect as we don't have to write tons of lines of code to do that, and we can just use our Laravel packages to charge the customer's credit card elegantly.

Next is to verify if payment is actually received. Here is the code that will verify payment and complete the order:

```
if($response->paid){
    $order_data = Order::find($order->original['id']);
    $order_data->transaction_id = $response->id;
    $order_data->status = 1;

    $order_data->save();

    $foldagram_data = Foldagram::find(Input::get('foldagram_id'));
    $foldagram_data->status = 2;
    $foldagram_data->user_id = $user;
    $foldagram_data->save();

    Cart->destroy();

    $userdata = Sentry::user(intval($user));
}
```

The preceding code will verify that the transaction is complete and it stores the transaction details in the order table. We are also updating the Foldagram status as Jordan needs new orders which are paid for importing them and sending Foldagram's via backend. At last, we are deleting the cart as the order is completed.

Building the credits section

Now it's time to work on the credits section. We would need two tables to manage the credits. One to manage credits pricing and one to manage the user's credit. We would need to create a migration first for our credits tables. First let's create the credit price table migration and run `php artisan migrate`:

```
php artisan migrate:make create_credits_table
```

It will generate the credits table basic schema update with the following schema:

```
class createCreditTable {

    public function up()
```

```
{
    Schema::create('credit', function($table)
    {
        $table->increments('id');
        $table->string("rfrom",255);
        $table->string("rto",255);
        $table->string("price",255);
        $table->timestamps();

    });
}
```

```
public function down()
{
    Schema::drop('credit');
}
```

```
}
```

Next, we need to create migration for the user's credit. Let's do that by running the following migration command:

php artisan migrate:make create_user_credit_orders_table

Also run `php artisan migrate` that will generate the schema file in the `app/database` directory, update it with the following schema:

```
class Create_User_Credit_Order_Table {

    public function up()
    {
        Schema::create('usercreditorders', function($table)
        {
            $table->increments('id');
            $table->integer('user_id');
            $table->integer('qty');
            $table->integer('price');
            $table->string("email",255);
            $table->boolean('status', array('0', '1'))->default(0);
            $table->string("transection_id",255);
            $table->timestamps();
        });
    }

    public function down()
```

```

    {
        Schema::drop('usercreditorders');
    }

}

```

Now we need to create models for these two tables in the `models` directory. So create `credit.php` and `usercreditorders.php` in the `models` directory and write the following code:

```

class Credit extends Eloquent {

    public static $table = 'credit';

    public static $timestamps = true;

}

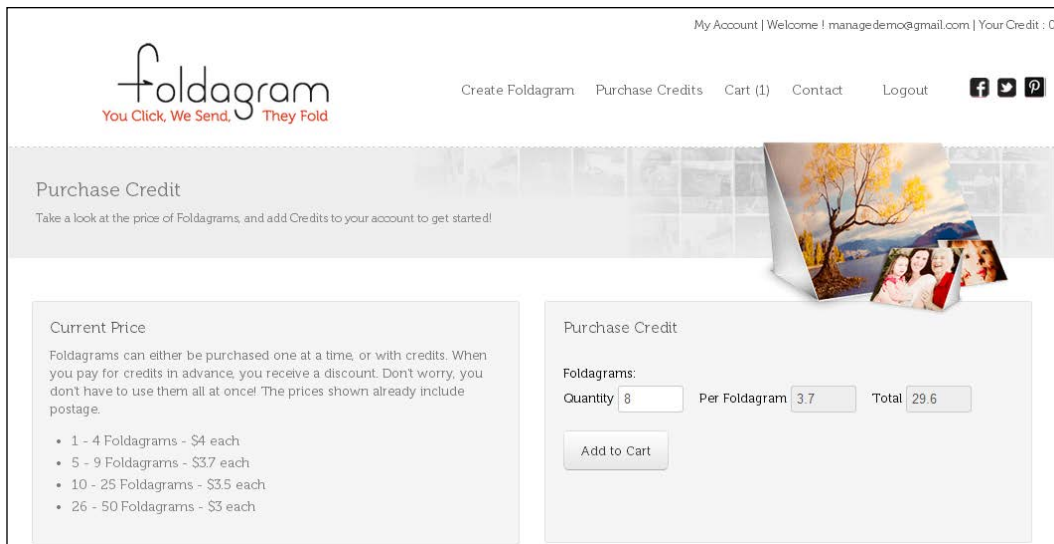
class UserCreditOrders extends Eloquent {

    public static $table = 'usercreditorders';

}

```

Now as we have everything set up for credits. Let's work on the **Purchase Credit** page for users. Here is our layout for the **Purchase Credit** page:



Purchase Credit page

As you can see the user can decide how many credits he wants to buy and based on the quantity, he'll get a discount.

First let's build the Controller to display the layout. Here is the code for that:

```
function get_purchase_credit()
{
    $credit = Credit::all();

    return View::make("pages.purchase_credit")->with("title", "The
Foldagram - Purchase Credit")
->with("page_title", "Purchase Credit")->with('class', 'pcredit')
->with('credit', $credit);
}
```

Here we are getting the whole price range from the credits table and sending it to our view file. Let's build our `purchase_credit.blade.php` file in the `views/pages` directory.

Let's first display prices in the left section of the page:

```
<div class="row price">
    <div class="span6 well price_content">
        <h3>Current Price</h3>
        <p>Foldagrams can either be purchased one at a time, or with
credits. When you pay for credits in advance, you receive a discount.
Don't worry, you don't have to use them all at once! The prices shown
already include postage.</p>
        @if($credit)
            <ul>
                @foreach($credit as $value)
                    <li>{{ $value->rfrom . " - " . $value->rto . " Foldagrams -
$. $value->price." each" }}</li>
                @endforeach
            </ul>
        @endif
    </div>
```

The preceding code will fetch the credit values from the `credit` array we send using Controller and build the credits price range table as shown in the *Purchase Credit page* screenshot.

The preceding code will display the **Purchase Credit** form section. There is one little thing we still need to do. To update the price when user changes the quantity, we need to add a JavaScript code into our page. Here is code for that:

- [147] -

The preceding code will generate the page for the purchase of credits. Now when the user submits the form, we need to check the user's credit card, charge it with our Stripe API, and add the credits to the user's account. Here is code for that:

```
public function post_addtocredit()
{

    $credit = Credit::where('rfrom','<=',intval(Input::get('qty'))
        ->where('rto','>=',intval(Input::get('qty')))->order_
by('rfrom','DESC')->first();

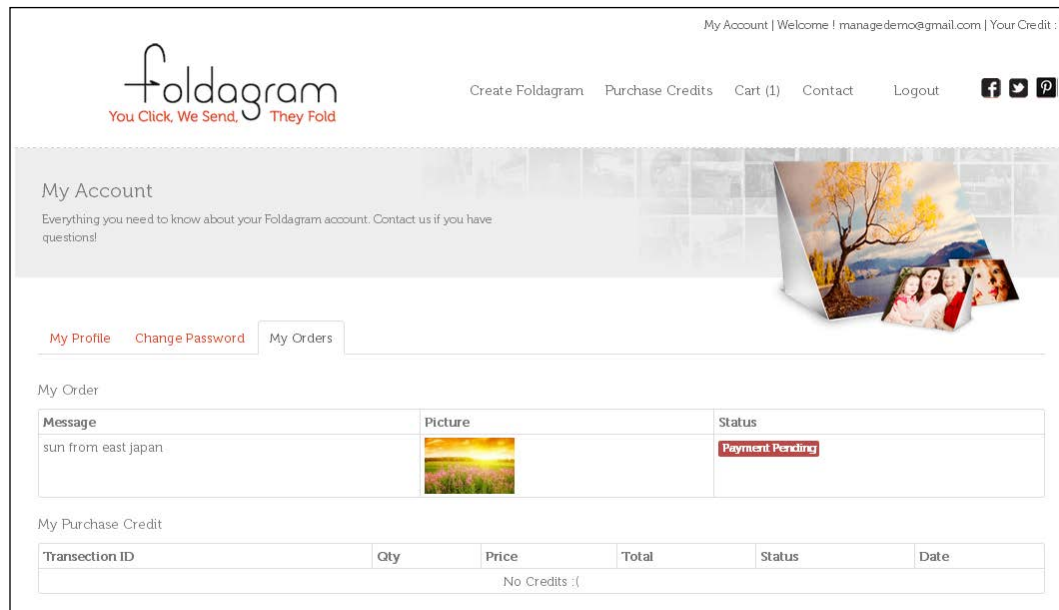
    $response = Stripe_Charge::create(array(
        "amount" =>round($credit->price * Input::get('qty')),
        "currency" => "usd",
        "card" => Input::get('stripeToken') , // obtained with Stripe.
js
        "description" => "Foldagram Payment")
    );

    if($response->paid){
        $user = Sentry::getUser();
        $order_data = new UserCreditOrders;
        $order_data->transection_id = $response->id;
        $order_data->status = 1;
        $order_data->user_id = $user->id;
        $order_data->qty = Input::get('qty');
        $order_data->price = $credit->price*Input::get('qty');
        $order_data->save();
    }
}
```

The preceding code will find the price from the price range based on quantity entered by the user. Then our Stripe package will charge the user's credit card based on the price found via qty*price range found from the credits table. If the transaction is successful, we are storing the credit quantity in the usercreditorders table.

Building the view orders section

As we have added functionality to pay via credit card or credits, now it's time to allow the users to view their orders and see the status of their orders. We need to display the user's orders in their dashboard screen. Here is the layout for the view orders from the designer:



User orders

Let's add a code to the third tab of views/user/dashboard.blade.php file. We already have the orders array passed to our dashboard View via our dashboard Controller method. So we can loop through orders easily.

```
<div class="tab-pane" id="tab3">
  <h3>My Order</h3>
  <table class="table table-striped table-bordered table-condensed ">
    <thead>
      <tr>
        <th>Message</th>
        <th>Picture</th>
        <th>Status</th>
      </tr>
```

```
</thead>
<tbody>
    @if(!empty($orders->results))
        @foreach($orders->results as $value)
            <tr>
                <td>{{ $value->message }}</td>
                <td>@if($value->image!="")

                    @if(File::exists(path('public').'img/
thumbnails/'.$value->image))
                        {{ HTML::image('img/thumbnails/'.$value-
>image, "Foldagram Image", array('width'=>'100px')) }}
                    @endif
                @endif

            </td>
            <td> <?php $status = Config::get('application.
status'); ?>

                @if($value->status=='1')
                    {{ Label::important($status[$value-
>status]); }}

                @elseif ($value->status=='2')
                    {{ Label::success($status[$value-
>status]); }}

                @elseif ($value->status=='3')
                    {{ Label::normal($status[$value-
>status]); }}

                @elseif ($value->status=='4')
                    {{ Label::warning($status[$value-
>status]); }}

                @elseif ($value->status=='5')
                    {{ Label::success($status[$value-
>status]); }}

                @elseif ($value->status=='6')
                    {{ Label::success($status[$value-
>status]); }}

                @elseif ($value->status=='7')
                    {{ Label::success($status[$value-
>status]); }}

                @endif
            </td>
        </tr>
```

```
        @endforeach
    @else
        <tr>
            <td colspan="3" class="sr-align-center" style="text-align: center">There is any order</td>
        </tr>
    @endif
</tbody>
</table>
```

Here in the preceding code, we are looping through the orders array and based on the status stored in our database, we are displaying different labels to the users about the order's current status.

Summary

So, we learned how we can manage user-related features in Laravel-based web applications. We have learned how to use Sentry to authenticate users and how to build sections like user login, user registration, user dashboard, user change password, and manage profile. We've also learned how we can build the checkout process and payment gateway integration with web applications. We built the following sections:

- Checkout process
- Pay via credit card
 - Integrated Stripe payment gateway
- Pay via credits
 - Pre payments for bulk orders
- View Orders

We have learned how to craft the user section in our Laravel 4 application. We have also learned how to use packages from `packagist.com` and use it to a great extent. Also we have built the frontend of our Foldagram web application. It's time now to move to the backend section of our web application.

7

The Admin Section

We finished building the frontend for `thefoldagram.com` in the previous three chapters. We learned how to use Laravel to create a web application that handles features such as cart, user credits, and orders. We created our own package for managing the cart functionality and built pages for the user's dashboard, login, and registration.

Now, we will learn how to build the administration area of our application. We will build functionalities such as managing orders and user management, which will be very useful for our friend Jordan. But first, it's time to meet him again and show him a demo of the different functions and discuss what he thinks about the admin section. So we met again, and after discussing the site, he told me that he would need to be able to do the following things as an administrator.

- Log in via a secure web page for the administrative area that normal users cannot login to
- See the orders placed in the application
- Update the order in case something requires to be changed
- Export the new orders to an Excel file
- Manage the price of a Foldagram unit
- Manage the price range of Foldagram units based on the quantity of the user credits
- Add credit for users
- Add new users
- Manage users, that is, update or block or delete them
- Change user passwords in case somebody requests a reset

This is the requirement for our second phase of iteration. So let's write down our requirements as features and try to get an idea of what we would need to accomplish them:

- Building the foundation for the administration section
- A login section for administrators
- Managing orders
- Exporting orders
- Managing Foldagram pricing
- Adding credit for user
- Managing users
- Resetting passwords
- Blocking users

Building the foundation for the administration section

To manage the backend section, we will create a separate layout with the admin Controller to hold the key to the backend. All routes of the administration section must start with `/admin` as that way we can manage things such as authentication and routes.

Before setting, let's discuss how we will differentiate normal users from administrators. Well, we will use the groups feature of **Sentry**, our authentication package. The way it works is that you set up groups in Sentry and then you add a user into a particular group; when whoever first accesses the routes you put a check on who belongs to that group.

Here we need to define an administration group. We will try out this procedure only once, so I will write a route closure function to achieve it:

```
Route::get('/creategroup', function()
{
    try
    {
        // Create the group
        $group = Sentry::createGroup(array(
            'name' => 'Administration',
            'permissions' => array(
                'read' => 1,
                'write' => 1,
```

```
        ),
    ));
}
catch (Cartalyst\Sentry\Groups\NameRequiredException $e)
{
    echo 'Name field is required';
}
catch (Cartalyst\Sentry\Groups\GroupExistsException $e)
{
    echo 'Group already exists';
}
});
```

Here, when you run `yoursite.com/creategroup`, Sentry creates the group named `administration` with two types of permission: `read` and `write`. But what are permissions? Well, permissions could be anything. Here, in the context of these chapters, it's for future requirements. For example, if we want to create an administrator who can only read data from it, we can set those permissions easily via Sentry.

Once the `administration` group is created, you can safely remove the route closure for security reasons. If you want to confirm whether the group is created, you can do so by running following code:

```
Route::get('/checkgroup', function()
{
    try
    {
        $group = Sentry::findGroupByName('administration');
    }
    catch (Cartalyst\Sentry\Groups\GroupNotFoundException $e)
    {
        echo 'Group was not found.';
    }
});
```

When you run `yoursite.com/checkgroup`, Sentry will try to search for the `administration` group in its group's tables. If Sentry doesn't find the group, it will return an error. That way, you will know your group has been created. Don't forget to delete this closure too.

Now you must be wondering about how to create a user that belongs to this group. Here is how you can do that:

```
Route::get('/createadminuser', function()
{

    try
    {

        // Find the group using the group name
        $adminGroup = Sentry::findGroupByName("administration");

        // Create the user
        $user = Sentry::createUser(array(
            'email' => 'adminuser@example.com',
            'password' => '123456',
        ));

        // Assign the group to the user
        $user->addGroup($adminGroup);
    }
    catch (Cartalyst\Sentry\Users>PasswordRequiredException $e)
    {
        echo 'Password field is required.';
    }
    catch (Cartalyst\Sentry\Users\UserExistsException $e)
    {
        echo 'User with this login already exists.';
    }
    catch (Cartalyst\Sentry\Groups\GroupNotFoundException $e)
    {
        echo 'Group was not found.';
    }

});
```

We first need to create the user via the `Sentry::createUser` method. We will then have a `User` object. After that, we can find the administration group using the `Sentry::findGroupByName` method. Now, via the `User` object, we have acquired that we created a user object having an `addGroup` method which allows us to add users to the group by just passing our object. If all goes well, Sentry will create a user `adminuser@example.com` and add it into the administration group of our application.

What if something goes wrong? Sentry will handle the error part via its exceptions. Say for example, a user with that e-mail address already exists in our database; Sentry will raise a `UserExistsException` exception. Similarly, Sentry will handle all the validations and we don't have to manually check any validations for any groups or users. Again, for obvious security reasons, don't forget to delete these routes after creating the administration user.

So now we have a theoretical administration group that will handle the administration functions of the application. We have created the administrator for our application too. Let's start working on the structure of the administration application.

To authenticate each admin request, we will create a route group. But what is a route group? Laravel provides a feature to add filters to a group of routes; this is called **group routing**. In group routing, you can define which filter should run before any route defined in that group is executed.

So we can write a filter that will authenticate the admin user and then use that filter in the route groups so that the user can be authenticated before any route is executed. Let's first define our administration filter in our `filters.php` file:

```
Route::filter('administrationauth', function()
{
    if (Sentry::check())
    {
        $user = Sentry::getUser();
        $admin = Sentry::findGroupByName('administration');

        if (!$user->inGroup($admin))
        {
            Redirect::to('login');
        }
    }
    else{
        Redirect::to('login');
    }
});
```

Here we first check whether a user is logged in or not via the `Sentry::check` method. If not, it will redirect the user to the login route. If the user is logged in, the `Sentry::getUser` method will return the currently logged in user as an object. Then we check via the `User` object whether the user is in the administration group or not. If they aren't, we redirect them to the login page. This filter will manage the whole administration authentication part for us.

Creating a login section for the administrator

Now as we have the basic code for managing the admin section, let's create a login section for the site. But first, we need to set up the layout for the admin section. So let's do that first. First, add a route to our login page as follows:

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

Now let's add our admin Controller code with the login method:

```
class AdminController extends BaseController {

    public function getLogin()
    {
        return View::make('layouts.login')->with('title','Foldagram -
Admin');
    }
}
```

Here, we are loading our login view from the layouts directory. So let's add the login layout file in our Views directory:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>{{ $title }}</title>
    <?php
        HTML::style('admin/css/bootstrap.css');
        HTML::style('admin/css/style.css');
        HTML::script('admin/js/jquery.js');
    ?>
</head>
<body class="body">
    <div class="navbar navbar-fixed-top">
        <div class="navbar-inner">
            <div class="container">
                <a href="#" class="brand">The Foldagram</a>
            </div>
        </div>
    </div>
```

The preceding code is the header of the login layout file. Note how we are using `HTML::style` and `HTML::script` to add CSS and JS from the `public/admin/css` and `public/admin/js` files. Then we will display the Foldagram logo as shown in the following code snippet:

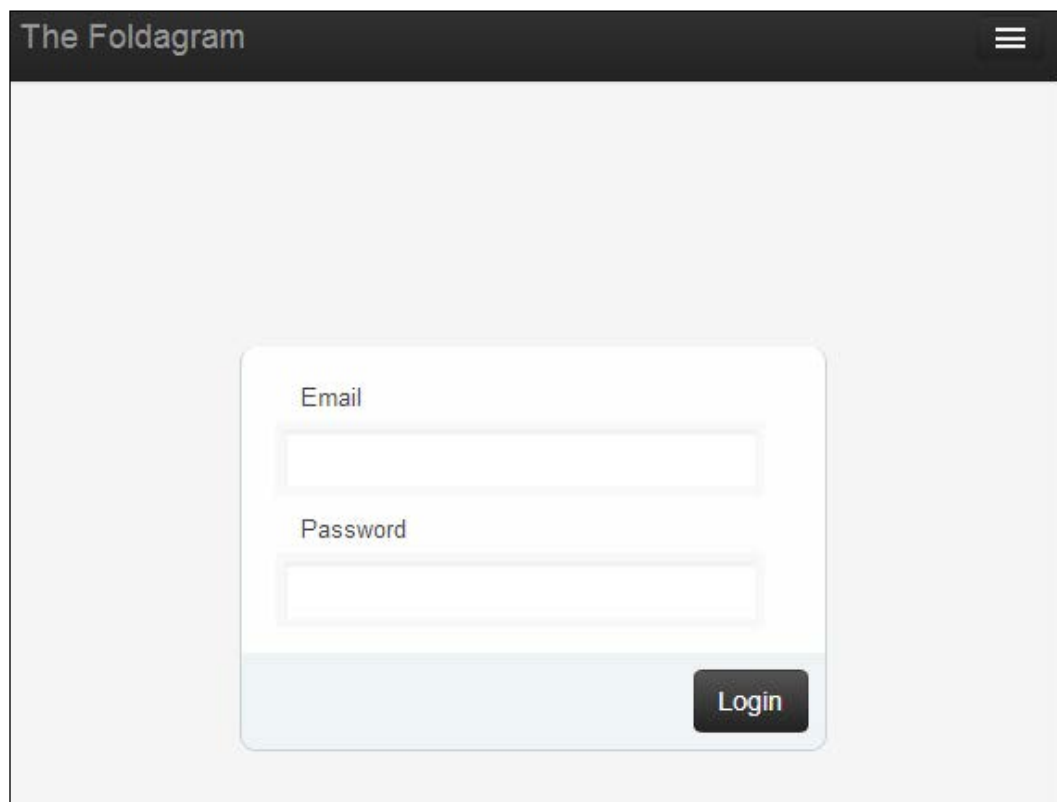
```
<div class="container-fluid sr-container">
  <div class="row-fluid ">
    @if($errors->has())
      <div class="alert alert-error alert-block">
        <button type="button" class="close" data-dismiss="alert">&times;</button>
        <h4>Error!</h4>
        <p>The following errors have occurred:</p>
        <ul id="form-errors">
          @foreach ($errors->all('<li>:message</li>') as $error)
            {{ $error }}
          @endforeach
        </ul>
      </div>
    @endif
```

The next section of code in login layout is for errors. If the validation sends back errors, we can retrieve it via the `$errors` object. So, the preceding code will loop through each error message for each field and display the appropriate error.

```
<div class="span12">
  <div class="content-container">
    {{ Form::open( array('url'=> 'admin/login')) }}
    <div class="container">
      <div class="content">
        <div class="box login">
          <fieldset class="boxBody">
            <label>Email</label>
            <input type="text" name="username"
              tabindex="1" value="{{
                Input::old('username')}} "
              placeholder="">
            <label>Password</label>
            <input type="password" name="password"
              tabindex="2" placeholder="">
          </fieldset>
          <div class="footer">
            <input type="submit" class="btn btn-large
              btn-inverse" value="Login" tabindex="3">
          </div>
```

```
        </div>
      </div>
    </div>
    {{Form::close()}}
  </div>
</div>
```

Finally, we have our login form. We are using the `Form` class to send user requests to the `admin/login` route via the `POST` method. Now try to check the URL at `http://yoursite.com/admin/login`; if all goes right, you will see the following screen:

The screenshot shows a web browser window with a dark header bar. On the left of the header is the text 'The Foldagram' and on the right is a hamburger menu icon. The main content area has a light gray background. In the center, there is a white login form with rounded corners and a subtle shadow. The form contains two input fields: the top one is labeled 'Email' and the bottom one is labeled 'Password'. Below these fields is a dark gray button with the word 'Login' in white text. The form is set against a light blue gradient bar at its base.

Admin login

Now we need to authenticate the user. Let's first add a `post` route and send a user request to our `postlogin` method of the `admin Controller`:

```
Route::post('admin/login', array('as' => 'adminlogin', 'uses' =>
    'admin@login'));
```

Now, in the admin Controller, we will write `postLogin`, which will authenticate the user. Here is our `postLogin` method code:

```
public function postLogin()
{

    $rules = array(
        'username' => 'required|max:50',
        'password' => 'required',
    );

    $input = Input::get();
    $validation = Validator::make($input, $rules);

    if ($validation->fails())
    {
        return Redirect::to_route('admin/login')->with_
errors($validation->errors)->with_input()->with("title", "The Foldagram
- Admin ");
    }
}
```

In the first section of the `postLogin` method, we are validating code make sure the user has filled the login boxes. Next we authenticate the user via Sentry:

```
try
{
    $user = Sentry::authenticate(Input::all(), false);
    if (Sentry::check())
    {
        $admin = Sentry::findGroupByName('Administration');
        if ($user->inGroup($admin))
        {
            return Redirect::route('orders');
        }
    }
}
```

Here we first use Sentry to authenticate the user. Then we check whether the user is authenticated using the `check` method in Sentry. And lastly, we will make sure the user is a member of the administration group. If he is a member of this group, we will redirect him to the orders page.

But what if he is not a member of the group or his password is simply wrong? Well, here is the exception code provided by Sentry; it will raise different types of exceptions and redirect the user to the login page.

```
catch (Cartalyst\Sentry\Users\WrongPasswordException $e)
{

    return Redirect::route('adminlogin', $id)
        ->withInput()
        ->with('message', 'Password field is required.');
```

```
}
catch (Cartalyst\Sentry\Users\UserNotFoundException $e)
{

    return Redirect::route('adminlogin', $id)
        ->withInput()
        ->with('message', 'User is not found in database.');
```

```
}
```

Managing orders

Now as we have authenticated the user, it's time to set up our dashboard and admin area. But first, let's write our group routes so that we don't have to check if user is admin each time we receive a request, and we can then just work on Controllers and Views:

```
Route::group(array('before' => 'administrationauth'), function()
{
    Route::get('admin/orders', array('as' => 'orders', 'uses' =>
'order@index'));
    Route::get('admin/user', array('as' => 'user', 'uses' => 'user@
index'));
    Route::get('admin/usercredit', array('as' => 'usercredit', 'uses'
=> 'credit@usercredit'));
    Route::post('admin/usercredit', array('as' => 'postusercredit',
'uses' => 'credit@usercredit'));
    Route::get('admin/credit', array('as' => 'credit', 'uses' =>
'credit@index'));
});
```

We have used `Route::group` to add a filter to all our admin section routes. Our `administrationauth` filter, which we created previously, is used to authenticate the user before the user can hit the route. This allows us to completely separate our authentication layer from our business logic. As you can see, we don't have to write any code further to authenticate the user as all routes in this group will first check the filter.

Let's get back to the creation of our manage orders section. We would need a Controller method for that. Here is method for that in the `orders` Controller:

```
public function getindex()
{
    $orders = DB::table('foldagram')
        ->left_join('users', 'foldagram.user_id', '=', 'users.id')
        ->left_join('foldagram_orders', 'foldagram.id', '=',
'foldagram_orders.foldaram_id')
        ->order_by('foldagram.created_at', 'DESC')
        ->group_by("foldagram.id")
        ->paginate(Config::get('application.pager'),
array('users.email',
    'foldagram.message',
    'foldagram.image',
    'foldagram.status',
    'foldagram.exported',
    'foldagram_orders.id',
    'foldagram.id AS fid',
    ));

    $pager = $orders->links(3, Paginator::ALIGN_LEFT,
Paginator::SIZE_SMALL);

    return View::make('admin.manage_orders')-
>with('title', 'Foldagram - Admin')
    ->with("page_title", "Manage Order")
    ->with('orders', $orders)
    ->with('pager', $pager);
}
```


Here, to display all orders, we use the query method of the database class to fetch all orders with user information. We are left joining our `Foldagram_orders` table with the user table to find the user information attached with each order. We use Laravel's paginator class to fetch records in sequence with page sets for paging. Then, we load View from the `order` layout file in the admin directory. Here is our `order` layout file:

```
@extends('layouts.admin')
@section('content')
{{ Form::open( array( 'url' => 'admin/orders/search')) }}

<label style="display: inline-block">Order Status :</label>&nbsp;
{{ Form::select('status', array('0'=>'-- Select Status
--')+Config::get('application.status'),Input::get('status'),
array('class'=>'input-medium')) }}

<input type="submit" name="submit" value="Search" class="btn btn-info
" style="display: inline-block">
{{ Form::close() }}
```

Here we first load our main template file from `layouts/admin.blade.php` and then override the content section with our order page content. We first allow the administrator to sort the orders via the status dropdown. Here we have all the common statuses from our config file.

Here is the status array that we load from the config file:

```
'status'=>array(
    '1'=>'Payment Pending',
    '2'=>'Order Accepted',
    '3'=>'Processing',
    '4'=>'Foldagram Ready',
    '5'=>'Shipping Pending',
    '6'=>'On Transit',
    '7'=>'Delivered',
);
```

Now when user changes their status and presses the search button, we will load the orders with the user's new status.

```
<table class="table table-striped table-bordered table-condensed">
    <thead>
        <tr>
            <th>Order No.</th>
            <th>Email</th>
            <th>Message</th>
```

```

        <th>Picture</th>
        <th>Status</th>
        <th>Export Status</th>
        <th>Action</th>
    </tr>
</thead>
<tbody>

```

Here we display the order column titles first. Now here's the remaining code:

```

        @foreach($orders->results as $value)
            <tr>
                <th>{{ $value->id }}</th>
                <td>{{ $value->email }}</td>
                <td>{{ $value->message }}</td>
                <td>@if($value->image!="")
                    @if(File::exists(path('public').'img/
thumbnails/'.$value->image))
                        {{ HTML::image('img/thumbnails/'.$value-
>image, "Foldagram Image", array('width'=>'100px')) }}
                    @else
                        {{ HTML::image ('img/thumbnails/100x100.
png') }}
                    @endif
                @endif
            </td>
            <td> <?php $status = Config::get('application.
status'); ?>
                {{ Label::important($status[$value->status]); }}
            </td>
            <td>
                @if($value->exported=="1")
                    {{ Label::success("Exported"); }}
                @else
                    {{ Label::normal("Not Exported"); }}
                @endif
            </td>
            <td>
                <a href="{{ { URL::to('order/recipient/'
. $value->fid) }}" class="btn"><i class="icon-pencil"></i> View
Recipient's</a>&nbsp;
                <a href="{{ { URL::to('order/orderdetail/' .
$value->fid) }}" class="btn btn-info"><i class="icon-pencil"></i>
Order Details</a>&nbsp;

```


Now we need to write four functions for each order so that the administrator can manage the orders easily. The four functions are as follows:

- View recipients
- Order details
- Update status
- Delete

Building the view recipients section

Administrator should be able to view recipients of orders via this feature. So let's write a Controller method for that.

```
public function getrecipient($id="")
{

    $reff = DB::table('foldagram_reff_address')->where('foldaram_
id','=', $id)->paginate(Config::get('application.pager'));

    $pager = $reff->links(3, Paginator::ALIGN_LEFT, Paginator::SIZE_
SMALL);

    return View::make('admin.view_recipeint')->with('title', 'Foldagram
- Admin')
->with("page_title", "List of Recipient's")
->with('reff', $reff)
->with('pager', $pager);
}
```

We are running a query to find the recipient of the Foldagram by querying the `Foldagram_ref_address` table. We are using the `paginator` class to generate pages for the records. Let's write a View code for displaying query results.

```
@extends('layouts.admin')
@section('content')
<h3>Foldagram Recipient's Address</h3>
<table class="table table-striped table-bordered table-condensed
sr-table">
    <tbody>
        @if(!empty($reff->results))
            <?php $i = 1; ?>
            @foreach($reff->results as $value)
                <tr>
                    <th>Recipient's Address {{ $i++ }} </th>
```

```
        </tr>
      <tr>
        <td>
          <address>
            <strong>{{ $value->fullname }}</strong><br/>
            {{ $value->address_one }}
          </address>
        </td>
      </tr>
    @endforeach
  @else
    <tr>
      <td class="sr-align-center">Record Not Found</td>
    </tr>
  @endif
</tbody>
</table>
{{ $pager }}
@end
```

The preceding code will display all recipients in tabular format for a particular order. We use `$reff` returned from Controller to generate results.

Building the order details section

The next feature we need to build is to display order-specific information such as transaction ID, quantity, price, and user information. Let's write a Controller method for that.

```
public function getorderdetail($id)
{
    $order_detail = Orders::where('foldaram_id','=', $id)->first();

    return View::make('admin.view_order_details')-
    >with('title','Foldagram - Admin')
    ->with("page_title","Order Details")
    ->with('order_detail', $order_detail->original);
}
```

Here we are using our Eloquent model orders to fetch the records based on the Foldagram ID and sending our Eloquent object to our `view_order_details` file.

Here is the `view_order_details` file in the `views/admin` directory to display order detail:

```
@extends('layouts.admin')
@section('content')
<h3>Foldagram Order Details </h3>
<table class="table table-striped table-bordered table-condensed
-table">
  <tbody>
    @if(!empty($order_detail))
      <tr>
        <th>Transection ID :</th>
        <td>{{ $order_detail->transection_id }} </td>
      </tr>
      <tr>
        <th>Quantity :</th>
        <td>{{ $order_detail->qty }} </td>
      </tr>
      <tr>
        <th>Price :</th>
        <td>{{ $order_detail->price }} </td>
      </tr>
      <tr>
        <th>Discount :</th>
        <td>{{ $order_detail->discount_amount }} </td>
      </tr>
      <tr>
        <th>Total :</th>
        <td>{{ ($order_detail->qty * $order_detail->price)
- $order_detail->discount_amount }} </td>
      </tr>
      <tr>
        <th>Email :</th>
        <td>{{ $order_detail->email }} </td>
      </tr>
      <tr>
        <th>Full Name :</th>
        <td>{{ $order_detail->fullname }} </td>
      </tr>
    </tbody>
  </table>
@stop
```

Here we display data from the `$order_detail` Eloquent object we got from the Controller. We calculate the total based on quantity. Here is what it looks like:

The Foldagram

Order Details

Back

Foldagram Order Details

Transection ID :	Purchase by credit -24
Quantity :	1
Price :	4
Discount :	
Total :	4
Discount Coupan Code :	
Email :	info@thefoldagram.com
Full Name :	test
Country :	def
Address One :	24,abc apt
Address Two :	
City :	rajkot
State :	gujarat
Zipcode :	360005

Order details page

Updating order status

Our next task is to build the update status page. We need to allow the administrator to change the status of the order. Let's build a Controller and View for that. Here is our Controller for getting the update status screen:

```
public function getupdate($id)
{
    if (!empty($id)) {

        $foldagram = Foldagram::find($id);

        return View::make('admin.update_status') -
        >with('title','Foldagram - Admin')
```

```

->with("page_title","Foldagram Update Status")
->with('foldagram',(object) $foldagram->original);

    }else{
        Redirect::to('admin');
    }
}

```

We are loading the Foldagram Eloquent object with the Foldagram ID and sending it to our View. Let's create the View file at `/views/admin/update_status`.

```

@extends('layouts.admin')
@section('content')
    <div class="span12 dcontent">
        {{ Form::open( array('url' =>'order/update')); }}
        <input type="hidden" name="id" value="{{ $foldagram->id }}">
        <div class="control-group">
            <label class="control-label" for="rfrom">Message :</label>
            <div class="controls">
                {{ $foldagram->message }}
            </div>
        </div>
        <div class="control-group">
            <label class="control-label" for="rto">Picture :</label>
            <div class="controls">
                @if($foldagram->image!="")
                    {{ HTML::image('img/thumbnails/'.$foldagram->image,
"Foldagram Image", array('width'=>'100px')) }}
                </div>
            </div>
            <div class="control-group">
                <label class="control-label" for="rto">Current Status :</
label>
                <div class="controls">
                    <?php $status = Config::get('application.status'); ?>
                    {{ Label::important($status[$foldagram->status]); }}
                </div>
            </div>
            <div class="control-group">
                <label class="control-label" for="rto">Change Status :</label>
                <div class="controls">
                    {{ Form::select('status', array('0'=>'-- Select Status
--')+Config::get('application.status'),$foldagram->status,
array('class'=>'input-medium')) }}
                </div>
            </div>
        </div>
    </div>

```




```
</div>
<div class="form-actions">
    <button type="submit" class="btn btn-primary">Save</button>
    {{ HTML::link_to_route('orders', 'Cancel', '',
array('class'=>'btn')) }}
</div>

{{ Form::close() }}
</div>
@stop
```

Here we display data from Foldagram into form inputs and are fetching status from the config file and setting it up via Foldagram's current status. If the admin wants to change the status he can do so by changing the value from the dropdown. Here is how our View file will be displayed:

Foldagram Update Status

Message : test

Picture : 

Current Status : Order Accepted

Change Status :

Order Accepted

-- Select Status --

Payment Pending

Order Accepted

Processing

Foldagram Ready

Shipping Pending

On Transit

Delivered

Updating order status

We would still need to write a method so that when the admin submits the updated status it will be saved. Here is a Controller method that will update the order status:

```
public function postUpdate()
{
    $foldagram = Foldagram::find(Input::get('id'));
    $foldagram->status=Input::get('status');

    if($foldagram->save()){
        return Redirect::to_route('orders')->with('success',
        'Foldagram order has been status updated successfully.');
```

Deleting orders

The last thing we need to do in the manage order page is to write a function that will allow the admins to delete the order. Generally, I don't allow deleting things unless it's absolutely necessary and because if you remove records that have many references, there will be a lot of errors in related pages. Also, sometimes a vital piece of information can be deleted by mistake.

But in this case, Jordan wanted this feature to remove dummy orders or spam orders. He had no intention of deleting any order that is paid for by the user. We agreed that we will remove the record from the database. Here is the code for that:

```
public function getdelete($id="")
{
    if(!empty($id)){
        $foldagram = Foldagram::find($id)->delete();
        $raddress = Raddress::where('foldaram_id','=', $id)-
>delete();
        $order_detail = Orders::where('foldaram_id','=', $id)-
>delete();

        if($foldagram){
            return Redirect::to_route('orders')->with('success',
            'Foldagram order has been deleted successfully.');
```

```
        }

        }else {
            return Redirect::to_route('orders')->with('error',"order
not deleted, please try again");
        }
    }
}
```

Here we are checking if the ID given already exists in the database and we are deleting records from Foldagram, reference address and order detail page via Eloquent's delete method. That completes our manage orders page functionality list.

Exporting orders

The next task in our queue is to allow Jordan to export new orders that are paid in the Excel format so that he can print Foldagram easily as per new orders.

```
public function get_csvexport()
{

    $output = Foldagram::with('foldagram_reff_address')
        ->where('foldagram_orders.status','=',1)
        ->where('foldagram.status','=',2)
        ->where('foldagram.exported','=',0)->get();

    $file = fopen('dummy/path/file.csv', 'w');
    foreach ($output as $row) {
        fputcsv($file, $row->to_array());
    }
    fclose($file);

    $headers = array(
        'Content-Type' => 'text/csv',
        'Content-Disposition' => 'attachment; filename="'.$file"',
    );

    return Response::make($file, 200, $headers);
}
```

The preceding code will first gather data via Eloquent loading method and save data into the file temporarily. Then admin can download the file as we are passing the download headers within the response.

Managing Foldagram pricing

The next functionality we need to build for the administrator is to manage the pricing per Foldagram based on range. Jordan wants to give discounts to users who buy credits (buy credits and prepay for Foldagram). Say if you are buying seven credits, you will pay less as the Foldagram price will depend on the range as follows:

- 1 to 4 Foldagrams: 4 USD/Foldagram or per credit
- 5 to 10 Foldagrams: 3.5 USD/Foldagram or per credit
- 11 to 20 Foldagrams: 3 USD/Foldagram or per credit

We will need to build an interface that will allow our admin to add price and also manage pricing. Let's start building an interface for to add pricing.

The screenshot shows a web application interface titled "The Foldagram". Below the title bar is a header section with the text "Add Price" and a "Back" button. The main content area contains three input fields labeled "From Qty:", "To Qty:", and "Price:". At the bottom of the form are two buttons: "Save" (highlighted in blue) and "Cancel".

Adding pricing

As you can see, the interface is quite simple; we just need to ask admin about the "from" quantity and the "to" quantity and the price for that range. Here is the Controller and View file for generating the form:

```
public function getadd()
{

    return View::make('admin.add_credit') -
        >with('title','Foldagram - Admin')
        ->with("page_title","Add Price");
}
```

Let's write a View file at views/admin/add_credit:

```
{{ Form::open( array ( 'url' => 'admin/credit/add') ) }}
    <div class="control-group">
        <label class="control-label" for="rfrom">Form Qty :</label>
        <div class="controls">
            <input type="text" name="rfrom" id="rfrom" value="{{
Input::old('rfrom') }}" placeholder="Form Qty">
        </div>
    </div>
    <div class="control-group">
        <label class="control-label" for="rto">To Qty :</label>
        <div class="controls">
            <input type="text" name="rto" id="rto" value="{{
Input::old('rto') }}" placeholder="To Qty">
        </div>
    </div>
    <div class="control-group">
        <label class="control-label" for="inputEmail">Price :</label>
        <div class="controls">
            <input type="text" name="price" id="price" value="{{
Input::old('price') }}" placeholder="Price">
        </div>
    </div>

    <div class="form-actions">
        <button type="submit" class="btn btn-primary">Save</button>
        {{ link_to_route('credit', 'Cancel', '', array('class'=>'btn')) }}
    </div>

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

Here we generate a form with three fields which will be posted at the credit Controller's postadd method. Let's write some code to save those inputs into the database:

```
public function postadd()
{
    $credit = new Credit();

    $credit->rfrom = Input::get('rfrom');
    $credit->rto = Input::get('rto');
```

```
$credit->price = Input::get('price');

if($credit->save()){
    return Redirect::to_route('credit')->with('success', 'Price
has been added successfully.');
```

```
}else {
    return Redirect::to_route('addcredit')->with_errors($credit-
>errors)->with_input()->with("title","The Foldagram - Admin");
}
}
```

In the `postadd` method, we first retrieve values for all inputs and then save it to our credit Eloquent object via the `save` method. These values are fetched by the frontend wherever we display/use pricing of the Foldagram.

Adding credit for the user

Jordan said he wanted to give credits to some of his early users as well as buyers who would purchase Foldagram in large quantities. We need to create a form where the admin can select a user from the dropdown and add a number of credits to that account. When they have credits, they can buy Foldagrams with those credits without paying via credit card.

Let's write a Controller method to first fetch all users and render the View file:

```
public function getusercredit()
{
    $users = Sentry::user()->all();

    $userarray = array();
    foreach($users as $value){
        $userarray[$value['id']] = $value['email'];
    }

    return View::make('admin.user_credit')->with('title','Foldagram -
Admin')
->with("page_title","Give User Credit")
->with('users',$userarray)
;
}
```

Here we first create an array of all users with key as ID and value as e-mail so we can easily display dropdown in our View file with that information. So let's write our View file at `/views/admin/user_credit.blade.php`:

```
<div class="span12 dcontent">
    {{ Form::open( array( 'url'=>'admin/usercredit') ) }}

    <div class="control-group">
        <label class="control-label" for="rfrom">Select User :</label>
        <div class="controls">
            {{ Form::select('user_email', array(''=>"-- Select User
--")+ $users, Input::old('user_email')) }}
        </div>
    </div>
    <div class="control-group">
        <label class="control-label" for="rto">Credit :</label>
        <div class="controls">
            <input type="text" name="credit" id="credit" value="{{
Input::old('credit') }}" placeholder="">
        </div>
    </div>

    <div class="form-actions">
        <button type="submit" class="btn btn-primary">Save</button>
        {{ link_to_route('user', 'Cancel', '',
array('class'=>'btn')) }}
    </div>

    {{ Form::close() }}
</div>
```

Here we are using the form helper's `select` method to generate our dropdown with the user's e-mails. When the user submits this form, it will be posted in the `credit` Controller's `postusercredit` method:

```
public function postusercredit()
{
    $user = Sentry::findUserByLogin((Input::get('user_email')));
    $credit = $user['metadata']['credit'] + Input::get('credit');

    $update = $user->update(array(
        'metadata' => array(
            'credit' => $credit,
        )
    ));

    if ($update)
```

```

    {
        return Redirect::to('admin/user')->with('success','Hey! User
credit has been updated successfully');
    }
    else
    {
        return Redirect::to('admin/usercredit')->with('error','Hey!
User credit has been not updated successfully');
    }
}

```

Here we first fetch the user via Sentry's `FinduserbyLogin` method, which will find the user with their e-mail in our system. Then we update the user with their new credit balance, which is the total of sum of their last credit balance and new credit entered by the admin.

Managing users

The next section that we need to build now is managing users in our system. We will need to build an interface from where the admin can add, edit, or delete users.

We also need to build the following two features Jordan requested:

- Block user
- Change user password

What we want to do is manage all this user-related functionality via one screen so the admin can easily do all these tasks. Here is the layout we decided and found that it would be easier to work with:

The Foldagram Home Orders Manage Price Manage Discount Subscriber User Management Howdy, info@thefoldagram.com						
Manage User Add User						
First Name	Last Name	Email	Credit	Status	User Type	Action
test	test	test@gmail.com	0	Active	Normal User	Edit Delete Block Purchase Credit Order
test2	test2	admin@admin.com	1	Active	Normal User	Edit Delete Block Purchase Credit Order
test32	test32	test32@gmail.com	21	Active	Normal User	Edit Delete Block Purchase Credit Order
test3	test3	test3@gmail.com	5	Active	Normal User	Edit Delete Block Purchase Credit Order
test4	test4	test4@gmail.com	0	Active	Normal User	Edit Delete Block Purchase Credit Order
jordan	bundy	menocollege@yahoo.com	15	Active	Normal User	Edit Delete Block Purchase Credit Order
The Foldagram	Admin	info@thefoldagram.com	24	Active	Admin User	Edit Delete Block Purchase Credit Order
« Previous 1 2 3 4 Next »						

Managing users

In this one screen, the admin has all the functions such as block user or edit user and can change a user's password or other information if required. We also have an **Add User** button, which will allow a user to add a new user into the system.

To add routes, we will add a Resource Controller, which will respond to UserController requests:

```
Route::controller('users', 'UserController');
```

Let's first build our interface for managing users. We need to add the `getIndex` method in our user's controller.

```
public function getIndex()
{
    $users =DB::table('users')
    ->join('users_metadata', 'users.id', '=', 'users_metadata.user_
id')
    ->paginate(Config::get('application.pager'));

    $pager = $users->links(3, Paginator::ALIGN_LEFT, Paginator::SIZE_
SMALL);

    return View::make('admin.manage_user')->with('title', 'Foldagram -
Admin')
    ->with("page_title", "Manage User")
    ->with('users', $users)
    ->with('pager', $pager);
}
```

Here we fetch all users' records with all metadata and bind it with the pagination class's method `paginate` for generating pages. Let's build our `manage_user` View at `views/admin/manage_user.blade.php`:

```
@extends('layouts.admin')
@section('content')
    <table class="table table-striped table-bordered table-condensed sr-
table">
        <thead>
            <tr>
                <th>First Name</th>
                <th>Last Name</th>
                <th>Email</th>
                <th>Credit</th>
                <th>Status</th>
                <th>User Type</th>
                <th>Action</th>
```

```

    </tr>
</thead>
<tbody>
    @if(!empty($users->results))

```

In our header section of View, we are extending our admin template. Up next are the table columns, which are used for displaying a user's fields in columns as a title in the template. Now here is the second section of our user layout:

```

        @foreach($users->results as $value)
            <tr>
                <td>{{ $value->first_name }}</td>
                <td>{{ $value->last_name }}</td>
                <td>{{ $value->email }}</td>
                <td>{{ $value->credit }}</td>
                <td>@if($value->status=="1")
                    <span class="label label-success">Active</span>
                @endif
                @if($value->status=="0")
                    <span class="label label-
important">Block</span>
                @endif
            </td>
            <td><a href="{ { URL::to('user/edit/' . $value->id)
}}" class="btn"><i class="icon-pencil"></i> Edit</a>
                <a href="{ { URL::to('user/delete/' . $value-
>id) }}" class="btn btn-danger"><i class="icon-remove-sign"></i>
Delete</a>
                @if($value->status=="1")
                    <a href="{ { URL::to('user/block/' .
$value->id) }}" class="btn btn-danger"><i class="icon-ban-circle"></i>
Block</a>
                @endif
                @if($value->status=="0")
                    <a href="{ { URL::to('user/active/' .
$value->id) }}" class="btn btn-success"><i class="icon-ok-circle"></i>
Active</a>
                @endif
                <a href="{ { URL::to('usercreditorder/' .
$value->id) }}" class="btn btn-info"><i class="icon-list"></i>
Purchase Credit Order</a>
            </td>
        </tr>
    @endforeach

```

```
        @else
            <tr>
                <td colspan="6" class="sr-align-center">No Users
Found</td>
            </tr>
        @endif
    </tbody>
</table>
{{ $pager }}
@stop
```

The preceding code block loops through users found via our `$users` Eloquent object. In each column, it displays the user's information. We generate links for each of our sections such as edit, delete, and block via the `URL::to` method.

Adding users

Let's build the other part of user management code one by one. First is the user section. Via this section, we can add users to the site directly as an admin. As always, we would need to add the Controller and View file first. Let's create our Controller method in the user Controller:

```
public function getadd()
{

    return View::make('admin.add_user')->with('title','Foldagram -
Admin')->with("page_title","Add User");

}
```

Here we just render our View `add_user` via `View::make`. Let's write our View file at `views/admin/adduser`.

```
{{ Form::open( array( 'url' => 'admin/user/add' ) ) }}
<div class="control-group">
    <label class="control-label" for="first_name">First Name </label>
    <div class="controls">
        <input type="text" name="first_name" id="first_name" value="{{
Input::old('first_name') }}" placeholder="">
    </div>
</div>
<div class="control-group">
    <label class="control-label" for="last_name">Last Name </label>
    <div class="controls">
```

```

        <input type="text" name="last_name" id="last_name" value="{{
Input::old('last_name') }}" placeholder="">
    </div>
</div>
<div class="control-group">
    <label class="control-label" for="rto">Email :</label>
    <div class="controls">
        <input type="text" name="email" id="email" value="{{
Input::old('email') }}" placeholder="">
    </div>
</div>
<div class="control-group">
    <label class="control-label" for="password">Password :</label>
    <div class="controls">
        <input type="password" name="password" id="password" value="{{
Input::old('password') }}" placeholder="">
    </div>
</div>
<div class="control-group">
    <label class="control-label" for="password">Confirm Password :</
label>
    <div class="controls">
        <input type="password" name="password_confirmation"
id="password_confirmation" value="{{ Input::old('password_
confirmation') }}" placeholder="">
    </div>
</div>
<div class="form-actions">
    <button type="submit" class="btn btn-primary">Save</button>
    {{ HTML::link_to_route('user', 'Cancel', '',
array('class'=>'btn')) }}
</div>

{{ Form::close() }}

```

Here we use a form helper to post forms to the `adduser` method of our user Controller. Let's write a `post` method of the `adduser` object to save the user into the database:

```

public function postadd()
{
    $data = array();
    $rules = array(
        'email' => 'required|email|unique:users',
        'password' => 'required|confirmed',
        'password_confirmation' => 'required'
    );
}

```

```
    );

    $input = Input::get();
    $validation = Validator::make($input, $rules);

    if ($validation->fails()) {
        return Redirect::to('admin/user/add')->with_input()->with_
errors($validation);
    }
}
```

Here we first validate input and make sure all fields for creating users are there. If the validator object returns false, then we redirect the user to the `with_errors` form.

```
$rules = array(
    'email' => 'required|email',
    'password' => 'required',
);

$input = Input::get();
$validation = Validator::make($input, $rules);

if ($validation->fails()) {
    return Redirect::to('admin/user/add')->with_input()->with_
errors($validation);
}

$user = Sentry::createuser(array(
    'email' => Input::get('email'),
    'password' => Input::get('password'),
    'metadata' => array(
        'first_name' => Input::get('first_name'),
        'last_name' => Input::get('last_name'),
        'credit' => 0,
    )
));

return Redirect::to('admin/user')->with('success', 'Hey! User has been
added successfully');
```

The preceding code block will save the user with the information provided in the form. We are using `Sentry::createuser`.

Editing users

We will be editing users just like we added users. The only difference between these two sections is in the `postedituser` method. In this method, we will need to update the user via the `Sentry::Save` method. Here is the code for the `Sentry::Save` method as a boilerplate code for both add and edit would be the same:

```
$user = Sentry::findUserById($id);

$user->password = Input::get('password');
$user->metadata = array(
    'first_name' => Input::get('first_name'),
    'last_name'  => Input::get('last_name'),
    'credit'     => 0,
);

if ($user->save())
{
    return Redirect::to('admin/user')->with('success','Hey! User info
has been updated successfully');
}
```

Here we first use the `finduserbyid` method to find the user, update user info via the `$user` object, and save the user via the `$user->save` method.

Deleting users

To delete a user, we need to first find the user and then, via `Sentry`, remove the user from the database. Here is code for that:

```
public function get_delete($id=null)
{
    try
    {
        $user = Sentry::findUserById(1);
        $delete = $user->delete();

        if ($delete)
        {
            return Redirect::to_route('user')->with('success', 'User
has been deleted successfully.');
```

```
        return Redirect::to_route('user')->with('error',"User not
        deleted, please try again")->with("title","The Foldagram - Admin");
    }
}
```

Blocking users

The final feature we need to build is to block the user. It helps manage those users who post spam on forms. Let's write a block method in our user's Controller:

```
public function get_block($id=null)
{
    $throttle = Sentry::findThrottlerByUserId($id);
    $blocked = $throttle->suspend();

    if ($blocked)
    {
        return Redirect::to_route('user')->with('success', 'User has
        been blocked successfully.');
```

```
    }
    else
```

```
    {
```

```
        return Redirect::to_route('user')->with('error',"User not
        block, please try again")->with("title","The Foldagram - Admin");
    }
}
```

```
}
```

To block the user, we use the `Sentry::findthrottlerByUserId` method, which will fetch the user, and then we can use the `suspend` method in the object provided by Sentry. This will block the user from logging in for the next few hours or days based on the configuration value `SuspensionTime` provided in the Sentry configuration file.

Summary

In this chapter, we have learned how to create the admin dashboard for our applications. We have also learned the following topics:

- Building the administration section foundation
- Log in section for administrators
- Exporting and managing orders
- Managing Foldagram pricing
- Adding credit for the user
- Managing users
- Resetting passwords

We have completed our Foldagram web application with this chapter. In the next chapter, we will learn how to build the RESTful APIs with Laravel.

8

Building a RESTful API with Laravel – Store Locator

In this chapter we will learn how to create REST APIs in Laravel 4. We will create a simple store locator API that will have a RESTful backend for sending API results. We will also build a frontend that will show you how to utilize APIs created via Laravel 4.

In this chapter we will cover the following topics:

- REST and its basics
- A store locator's single page web application
- Creating a REST API in Laravel 4 using Resource Controllers
- Creating a RESTful backend
- Creating an API to view all the stores
- Building an API method for viewing an individual store
- Creating an API method for searching the stores
- Building an add store method of our API
- Creating an update store method of our API
- Creating an API method for deleting a store
- Creating a frontend via the Restful API

REST basics

First let's learn what REST is. **REST** stands for **Representational State Transfer**. It's an architectural style developed by **World Wide Web Consortium (W3C)** or a web model.

The REST architecture is designed using the following components:

- **Clients:** Clients initiate a request.
- **Servers:** Servers process and respond to requests.
- **Resource:** Resource is a concept to describe things based on the usage of your API. So if you are developing an API for an e-commerce product company, the product is the resource.

REST uses HTTP request methods to describe the API. Mixing resources with the HTTP request type gives you the REST architecture. Another thing that is required for handling data is the media type of the data. Mostly, **JSON** is used as the media type for REST APIs. Here is a sample architecture describing REST.

Assume that we are developing a sample architecture for managing people via the REST API for a government application. So `http://sample.com/API/v1/persons/` is the base URI for our API. Here API can be changed many times over a period of time, so versioning your API will help developers to know which features are available on which version. This is the reason why we will use `v1` in our API.

Here is a table describing some common functions that our API will provide:

Resource	POST (create)	GET (read)	PUT (update)	DELETE (delete)
/persons	Creates a new person	Lists all the persons	Updates all the persons	Deletes all the persons
/persons/1	Not used with this resource	Shows a person with the ID 1	Updates if a person with the ID 1 exists	Deletes the person with the ID 1

Here, as you can see, we would be able to manage our application via our resource persons easily. And these structures allow many departments of the government to work on the project later if they need to add, update, or delete data from other systems. So this is where the REST API shines and gives some great structure to work with projects where you don't know how the end point system will be developed.

A store locator's single page web application

We need to build an application for a multinational retail company having a lot of departments and resellers in different countries. The challenge is to establish a REST API structure that can be used by resellers or departments that have their web applications built in different platforms. So they can just call our REST APIs for having the APIs displayed in different retail stores.

We also need to build a frontend for the main site that will call our REST API and display all the nearby stores on the Google map. We would also need to create an authentic mechanism for departments that can add, edit, or delete stores within the API. We will use Laravel's Resource Controllers to create our API.

Creating a REST API in Laravel 4 using Resource Controllers

Resource Controllers are an easy way to create a REST API in Laravel 4 around its resources. Laravel 4 removes a lot of boilerplate code that we have to write, which we would have to write if we start our API from scratch. Let's generate the Resource Controller for our application. We need to create our API around the `stores` resource. So let's create the Resource Controller via the artisan command:

```
$ php artisan controller:make StoreController
```


This command will generate a Controller `StoreController.php` file at `app/controllers/`. If you check the file, you will find boilerplate Controller methods to use with the API. You will be thinking, "How does Laravel know about these files as we haven't declared routes for all these methods?" Here's another surprise; you just need one route line to use all these methods. The route declaration needed to register our Controller as Resource Controller is shown as follows:

```
Route::resource('store', 'StoreController');
```

`Route::resource` is the way we can register our Resource Controllers with Laravel. This single route line will handle all our REST API calls. Here is the list of routes it will handle for us:

HTTP Method	PATH	Action	Route Name
GET	/store	To view all the stores	store.index
GET	/store/create	To create a new store	store.create
POST	/store	To save new store	store.store
GET	/store/{id}	To show a store with {id}	store.show
GET	/store/{id}/edit	To edit the store	store.edit
PUT/PATCH	/store/{id}	To update the store	store.update
DELETE	/store/{id}	To destroy the store	store.destroy

Here the route name will match our Resource Controller's boilerplate methods. That's how Laravel manages routing automatically for all the routes.



If you want to check all your routes via the command line, artisans have a great command for that. Just type in the following command:

```
$ php artisan routes
```

The preceding command will display the list of all the routes registered within the application. It will also show you the hidden routes that Laravel manages in case of Resource Controllers.

Creating a RESTful backend

Now as we have set up our Resource Controller, we need to set up other backend options, such as database and authentication.

First we need to set up the database structure of our application. Here are the two tables required for doing this:

- `stores`: This table is to manage all the stores
- `users`: This table is to manage the users of an application

The `stores` table will have the following fields:

- `id`
- `name`

- address
- city
- postcode
- state
- country
- latitude
- longitude
- support_phone
- support_email

The users table will have the following fields:

- id
- username
- password

Let's write migrations to create tables in our database:

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

This will set up our schema files at `app/database/migrations`. First let's edit our stores migration file to add fields.

```
Schema::create('stores', function(Blueprint $table)
{
    $table->increments('id');
    $table->string('name');
    $table->string('address');
    $table->string('city');
    $table->string('zip');
    $table->string('state');
    $table->string('country');
    $table->string('latitude');
    $table->string('longitude');
    $table->string('support_phone');
    $table->string('support_email');
    $table->integer('user_id');
    $table->timestamps();
});
```

Now let's edit the users migration file to add users table's fields.

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

Let's run migrations so our artisan command can create our tables in the database:

```
$ php artisan migrate
```

We would also need to create a model for our stores table so we can access the table via the Eloquent object. Let's create a file at app/model/stores.php:

```
class Store extends Eloquent {

    protected $table = 'stores';

}
```

For the users table Laravel already provides us with model users. Laravel also provides us with the basic authentication, so let's use that to create a filter:

```
Route::filter('Apiauth', function()
{
    return Auth::basic("username");
});
```

To authenticate users we will use Laravel's route groups as well as prefix the API with a custom parameter. We would need to wrap our resource route with Route::group:

```
Route::group(array('prefix' => 'locator/api/service/v1', 'before'
=> 'Apiauth'), function()
{
    Route::resource('store', 'StoreController');
});
```

The previous code will allow us to authenticate each request to our API as we are using the before parameter with our filter to authenticate a user. We are using a prefix, so any requests that are made at `example.com/locator/api/service/v1` will hit our API. So we have everything ready to create our API. Let's create each option of our API one by one.

Creating an API to view all the stores

Let's create an API feature to view all the stores. As per the Resource Controller for all the stores, a request will hit the index method of `storecontroller.php`:

```
public function index()
{
    $stores = Stores::all();

    return Response::json(array(
        'error' => false,
        'stores' => $stores->toArray(),
        200
    )->setCallback(Input::get('callback')));
}
```

Here we are first fetching all the records from `stores` via the Eloquent object: `stores:all()`. Laravel has a unique method of sending a response as a JSON with the status and error code. `Response::Json` accepts two arguments: first is an array that shows the error code and the JSON object with data and the second argument is the status code of the request. The `setcallback()` method at the end allows us to send any custom callback requested by the client side. So this will return all the stores in the JSON format to the client.

Building an API method for viewing an individual store

To view an individual store, the request method will be the `show` method of the store Controller. Let's edit the `show` method to view the specific store:

```
public function show($id)
{
    $store = Stores::find($id);

    return Response::json(array(
        'error' => false,
        'store' => $store),
        200
    )->setCallback(Input::get('callback'));
}
```

Here, first we are fetching a record via the `id` parameter passed by the client and then sending it as a JSON response.

Creating an API method for searching the stores

The next option that we need to provide is that of searching the store. A client will provide us with search options such as city, country, and zip code, and we need to find it in our database and return all the nearby retail stores. Here the API call will be like `/stores?city=london`, but we will send our search data via the `get` method. So let's edit our `index` method as our API will call the `index` method:

```
public function index()
{
    $city = Input::get('city');
    $country = Input::get('country');
    $zip = Input::get('zip');

    If ( $city == '' && $country == '' && $zip == '' ){
        $stores = Store::all();
    }else{
        $stores = DB::table('store')
            ->where('city', 'LIKE', "%$city%")
            ->or_where('country', 'LIKE', "%$country%")
            ->or_where('zip', '=', "$zip")
            ->get();
    }

    return Response::json(array(
        'error' => false,
        'stores' => $stores->toArray()),
        200
    )->setCallback(Input::get('callback'));
}
```

Here first we are checking whether we have any inputs from the user. If there are no inputs, we are returning all the stores. If there are any inputs, we are using a fluent query builder to build the query via the `DB::table` method. After that we are using the response object to send data in the JSON format.

So we have created methods for reading from an API. Now let's write API methods that will allow a user to write data via the API.

Adding a store method to our API

To add a store, our Resource Controller route will call the store method. So let's edit the store method to save the new store entry given by the client:

```
public function store()
{
    $store = new store;
    $store->name = Request::get('name');
    $store->address = Request::get('address');
    $store->city = Request::get('city');
    $store->zip = Request::get('zip');
    $store->country = Request::get('country');
    $store->latitude = Request::get('latitude');
    $store->longitude = Request::get('longitude');
    $store->support_phone = Request::get('support_phone');
    $store->support_email = Request::get('support_email');
    $store->user_id = Auth::user()->id;

    if( $store->save() ){

        return Response::json(array(
            'error' => false,
            'msg' => 'store created successfully.',
            200
        )->setCallback(Input::get('callback')));
    }else{

        return Response::json( array(
            'error' => true,
            'msg' => 'issue creating store!'),
            200
        )->setCallback(Input::get('callback')));
    }
}
```

Here we are saving the store method first. If it's saved, we would send the success code with the success message via the `msg` variable; if it's not created, we would send an error. We are also saving the user ID with the current authenticated user's ID for managing who created the store.

Updating the store method of our API

To update the store method, we would need to change the update method as it is the method called by Laravel when either PUT or PATCH request is sent via the client. Let's write the update method to allow the store to be edited by the client:

```
public function update($id)
{
    $store = Store::find($id);

    $store->name = Request::get('name');
    $store->address = Request::get('address');
    $store->city = Request::get('city');
    $store->zip = Request::get('zip');
    $store->country = Request::get('country');
    $store->latitude = Request::get('latitude');
    $store->longitude = Request::get('longitude');
    $store->support_phone = Request::get('support_phone');
    $store->support_email = Request::get('support_email');
    $store->user_id = Auth::user()->id;

    if( $store->save() ){

        return Response::json(array(
            'error' => false,
            'msg' => 'store has been updated',
            200
        ))->setCallback(Input::get('callback'));
    }else{

        return Response::json( array(
            'error' => true,
            'msg' => 'issue updating store!'),
            200
        ))->setCallback(Input::get('callback'));

    }

}
```

To update the store, we first need to find the store from the database via the `Store::find` method. Then, we would use the properties of the Eloquent object to update individual properties of the record. If the record is updated, we would send the response with the success message or else the error message.

Creating an API method for deleting a store client

To delete a store client, we will send a request via the `delete` method and the request will have a store ID. Our resource stores will send a request to the `destroy` method. So let's update the `destroy` method to remove a store sent by the client.

```
public function destroy($id)
{
    $store = Store::find($id);
    if(!empty($store))
        $store->delete();

    return Response::json(array(
        'error' => false,
        'msg' => 'Store deleted'),
        200
    )->setCallback(Input::get('callback'));
}
```

Here we are first finding a store via the store Eloquent object's `find` method. Then via the Eloquent object's `delete` method, we are deleting the user.

So that's how you can write via an API. With Laravel 4 we have seen it's very easy to write APIs. As it provides a lot of boilerplate code as well as intelligent methods that we can use directly, it saves a lot of time and APIs are really elegant to work with.

You might be wondering how can we test all these APIs we created. We can test it in two ways:

- **Curl:** This is a command-line tool
- **POSTMAN:** This is a chrome extension that allows testing APIs even in local development environments

Let's test our API with curl. Say, if we want to retrieve all the stores, we will need to write the following command with curl:

```
$ curl --user test:test example.com/locator/api/service/v1/stores
```

If the username and password is correct, the preceding command will have the following output:

```
{
    "error": false,
    "stores": [
        {
```

```
        "id":1,
        "name":"NY store",
        "address":"times square",
        "city": "new york",
        "zip": "10032"
        "country": "usa",
        "latitude": "40.6700",
        "longitude": "73.9400",
        "created_at": "2013-02-01 02:39:10",
        "updated_at": "2013-02-01 02:39:10",
        "user_id": "1"
    },
    {
        "id":2,
        "name":"London store",
        "address":"30 Leicester Square",
        "city": "London",
        "zip": "WC2H 7LA"
        "country": "London",
        "latitude": "51.5072",
        "longitude": "0.1275",
        "created_at": "2013-02-01 02:39:10",
        "updated_at": "2013-02-01 02:39:10",
        "user_id": "1"
    }
]
}
```

Here if you observe the output, it has timestamps; if you don't want to show timestamps or any other column, you can insert a hidden property in your model as follows:

```
protected $hidden = array('created_at','updated_at');
```

For testing with **POSTMAN**, you will need to have a chrome browser. Once you open chrome, open the following link for chrome extensions:

- <https://chrome.google.com/webstore/category/extensions>

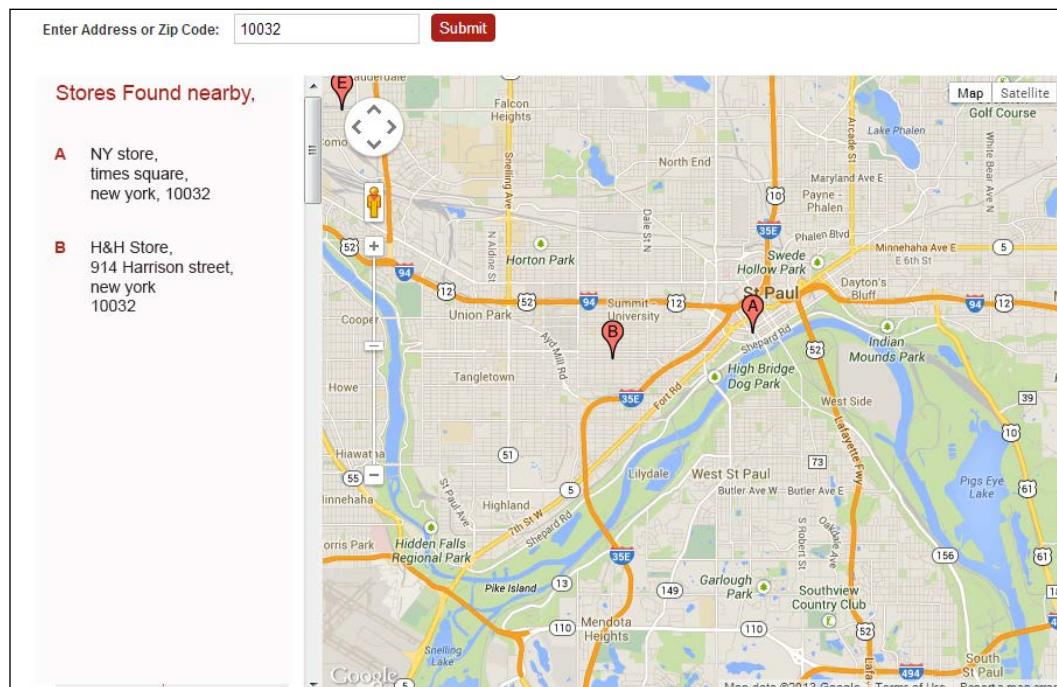
In the top-left box search for **Postman - REST Client** and click on the first result. A pop up will open with information about the extension. Click on the install button at the top-right corner of the pop up.

Once you have installed **POSTMAN** click on the second tab titled **basic auth**; enter the **URL**, **username**, and **password**; and click on the **send** button. It will display all the information about the request as well as the response in the same window.

So choose your favorite tool and start testing the whole API. We have finished creating our API's backend and it's time to move onto the frontend section. Let's learn how we can utilize REST APIs and create the store locator's frontend.

Creating a frontend via a RESTful API

Let's build a frontend for searching our stores on Google maps via our REST API. To set up the frontend, I have used jQuery and the Google Maps Store Locator plugin. Here is the preview page of our application:



Store locator – Google maps

Here is code for creating a layout similar to the preceding figure:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Store Locator</title>
    <meta charset="UTF-8">
    <link rel="stylesheet" type="text/css" href="css/map.css" />
```

```
</head>

<body>

  <div id="store-locator-container">

    <div id="form-container">
      <form id="user-location" method="post" action="#">
        <div id="form-input">
          <label for="address">Enter Address or Zip
            Code:</label>
          <input type="text" id="address" name="address" />
        </div>

          <button id="submit" type="submit">Submit</button>
        </form>
      </div>

      <div id="map-container">
        <div id="loc-list">
          <ul id="list"></ul>
        </div>
        <div id="map"></div>
      </div>
    </div>

    <script src="http://code.jquery.com/jquery-
      1.10.1.min.js"></script>
    <script src="js/handlebars-1.0.0.min.js"></script>
    <script
      src="http://maps.google.com/maps/api/js?
        sensor=false"></script>
    <script src="js/jquery.storelocator.js"></script>
    <script>
      $(function() {
        $('#map-container').storeLocator({'dataType': 'json',
          'dataLocation': 'example.com/locator/api
            /service/v1/stores.json'});
      });
    </script>

  </body>
</html>
```

Here we have a form that will ask the user for the zip code or the city name which will be submitted to our REST API. Then we have a map container that is initialized at the end of the page via the `storelocator` method. We are passing the JSON file that will be generated by our API we created in earlier parts of our application.

Here is some layout CSS code to make our page look like the layout:

```
#store-locator-container{
    float: left;
    margin-left: 20px;
    width: 875px;
    font: normal 12px Arial, Helvetica, sans-serif;
    color: #333;
}

#page-header{
    float: left;
}

#form-container{
    clear: left;
    float: left;
    margin-top: 15px;
    width: 100%;
}

#map-container{
    clear: left;
    float: left;
    margin-top: 27px;
    height: 530px;
    width: 875px;
}

#map-container a{
    color: #e76737;
    text-decoration: none;
}

#map-container a:hover, #map-container a:active{
    text-decoration: underline;
}

#map-container .custom-marker{
    width: 32px;
```



```
        height: 37px;
        color: #fff;
        background: url(../images/custom-marker.png) no-repeat;
        padding: 3px;
        cursor: pointer;
    }

    #loc-list .list-focus{
        border: 1px solid rgba(82,168,236,0.9);
        -moz-box-shadow: 0 0 8px rgba(82,168,236,0.7);
        -webkit-box-shadow: 0 0 8px rgba(82,168,236,0.7);
        box-shadow: 0 0 8px rgba(82,168,236,0.7);
        transition: border 0.2s linear 0s, box-shadow 0.2s linear 0s;
    }

    #map-container .loc-name{
        color: #AD2118;
        font-weight: bold;
    }
}
```

So this completes our frontend API code. We have learned how to use Laravel 4 to create simple APIs and how to authenticate users in API as well as how to send a JSON response to the client.

Summary

In this chapter we have learned how to build RESTful applications with Laravel 4. In the initial part of the chapter we learned the basics of REST and then we built a store locator's single page application.

Towards the end of this chapter, we learned how to code the RESTful backend and frontend via Laravel 4 and jQuery. In the next chapter, we will learn how we can use Laravel 4 to debug, optimize, and secure our applications.

9

Optimizing and Securing Our Applications

In this chapter we will learn how we can optimize Laravel via the profiler and how to secure our applications. We will also learn how to log data with Laravel as well as how we can add security layers to our application. Here are the topics we are going to cover in this chapter:

- Handling errors with Laravel
- Profiling Laravel applications
- Logging data with Laravel
- Security in Laravel
 - SQL injections
 - CSRF
 - XSS

We have learned how to create applications with Laravel 4 in prior chapters. Now, we need to focus on small issues and details that will help us optimize Laravel configurations. Let's start with errors.

Handling errors

By default, Laravel sets the debug option in `app/config/app.php` as `true`, so whenever an error occurs, Laravel shows a detailed error page. This will show all kinds of information about the error, such as the stack trace, the filenames, and the error line with reference to the code block. This is ideal when you are developing your application, but showing those pages to users in production will really scare them. So when you are in a production environment, you need to switch off the error variable by setting it to `false` in your app file.

Now the question is how to track errors in production. If we turn off the debug variable, how do we know whether there are any errors and how to handle these errors? Returning a blank page on error occurrence will not help users, and switching off the error will not help us debug it. Well, Laravel has an answer for that: log files. In your `logs` directory at `app/storage/`, you will have a log of errors on file for each day. So, if you want to find out what happened on your production site yesterday, just check the log file `log-<date>.txt` and you will have all the details you want, including the error stack and any common error occurring without your knowledge. Once you put your site into production, look for logs for a few days as there might be bugs or errors that you don't know and that are invisible to users.

The next question is how to we handle error pages. On error occurrence, you may want to send the user to a custom page on which they can see relevant pages or search pages. Sometimes, you may want to send them to your sitemap. So you will need to have control over the error pages. Guess what! You can do that easily with Laravel error handlers.

We want to manage a situation in which a certain page is not found and the user needs to be directed to a much more search friendly page with our sitemap. To do this you will need to add the following entry to your `global.php` file placed at `app/start/`:

```
App::missing(function($exception)
{
    return Response::view('custom404page', array(), 404);
});
```

When Laravel framework is initialized, our `global.php` file is scanned by the framework for any custom error handlers. Here in our case Laravel will pick that for any missing page (error code 404) and will load the `view` file defined in our route from `app/views/custom404page.blade.php`.

Laravel also provides an error handler that will catch any exception thrown from your application and show the custom page or error. Here is the custom exception handler format:

```
App::error(function(Exception $exception)
{
    // Handle the exception...
});
```

So how do we use this for something of our own? Remember we had cart exceptions in *Chapter 5, Creating a Cart Package for Our Application*. Say for example we want to show a custom error page for `CartInvalidDataException` to the user to tell them that there is some problem with the cart. How do we do that? It's simple; just use your exception in an error handler as shown in the following code snippet:

```
App::error(function(CartInvalidDataException $exception)
{
    Log::error($exception);
    return Response::view('Cart.dataerror');
});
```

Here, Laravel will pick our error handler, and whenever the error handler catches `CartInvalidDataException`, it will respond with `dataerror` in the `cart` folder of your `views` directory. Similarly, you can use other packages. Say for example you are using **Sentry**; you can use exceptions such as `UserSuspendedException` to display users who have suspended the custom page instead of the dashboard.

We have learned how to handle errors and give custom responses and handle custom exceptions. Let's learn how to profile our application and how to find out about what's happening on the page. In other words, we learn to find out information such as input variables (`$_GET` and `$_POST`) and all the queries run on the page or how much time it takes to execute a page or the amount of memory consumed for executing a given page.

Profiling Laravel applications

Profiling is measuring the software program's time or memory for a given function/class or any code block. The most common purpose of profiling applications is to aid application optimization.

There are two very good Laravel packages that will allow you to profile your applications:

- `juy/profiler`
- `loic-sharma/profiler`

Both are based on the Laravel 3 profiler and have almost the same features, but `juy/profiler` has a few more features and I use it for profiling my applications. Here is a list of things you can track with `juy/profiler`:

- Environment information
- Current controller/action info
- Routes

- Log events
- The SQL query log with syntax highlighting
- Total execution time
- Total memory usage
- All variables passed to views
- Session variables
- Sentry authentication variables

Isn't this awesome? When you are developing applications, you will have all this information right at your fingertips on each request. Thus you will instantly be able to tell when something goes wrong even if it may not show in the page output (but is obviously wrong in the background). This is because, you have created code that loops queries. This profiler will let you know you are running x queries at the bottom of the page and you would know instantly something is wrong with the page. You can then refactor your code to fix those mistakes. Let's see how we can install and use the profiler in our Laravel application.

To install the profiler go to your `composer.json` file and add following entry to the `require` array:

```
"juy/profiler" : "dev-master"
```

Then, run `composer update` so that Composer downloads the package and puts it in the vendor's directory. Now to hook it up with our Laravel application we will need to add its service provider into the `providers` array in the `app.php` file. Add the following line to the code in the `app.php` configuration file:

```
'Juy\Profiler\Providers\ProfilerServiceProvider',
```

Also add the following alias in the `alias` array in the `config` file:

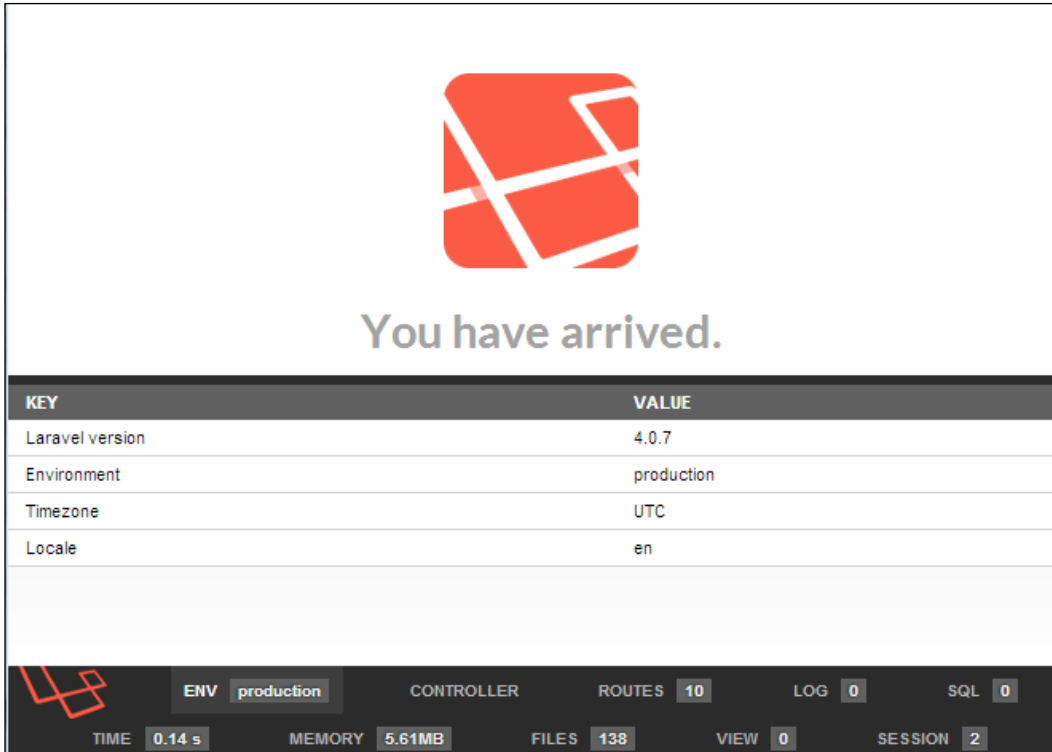
```
'Profiler' => 'Juy\Profiler\Facades\Profiler',
```

One last step is to publish the `config` files of the package so that our `config` file doesn't get overwritten if we update the package:


```
$ php artisan config:publish juy/profiler
```

Now you will have the `config` file available at `app/config/packages/juy/profiler/`. Just change the `profiler` option to `true`. And you will have a profiler bar at the bottom of your page in your browser. The profiler will display the information related to your page and environment in small tabs at the bottom of your page. For example, a SQL section will contain all the queries and an environment section will have information about which environment is loaded and which Laravel version you are using right now.

As you can see, it gives you a lot more information, such as memory allocated by current page and which files are used by the framework to generate this page as well as any session variables used in the page. Here is the preview of the juy profiler in action:



KEY	VALUE
Laravel version	4.0.7
Environment	production
Timezone	UTC
Locale	en


ENV **production**
CONTROLLER
ROUTES **10**
LOG **0**
SQL **0**

TIME **0.14 s**
MEMORY **5.61MB**
FILES **138**
VIEW **0**
SESSION **2**

Juy Profiler for Laravel

Another very important use of the profiler is to find out the time needed to execute any block of code in your application. Say for example you want to profile time between blocks of code; you can write the following code and the profiler will display the time needed to execute the code in the time tab:

```
Profiler::start('Blocka started');
DB::query('select * from auther where id= '.$id);
Profiler::end('Blocka completed');
```

In the preceding example, we can see the time between executing two simultaneous queries via our profiler. This is how you can optimize sections of your page by finding which section takes more time.

Logging data with Laravel

Another important aid to understanding why errors occur is the logging of important events during page execution. Laravel provides logging methods that we can use to save a lot of things that happened in the page. This is very useful in certain situations, such as When you are doing beta testing and users complain that certain pages or processes are not working; you can actually debug things via this data and can create a test case for finding errors that the users face. This can be done using the following code:

```
Log::debug('input data at pointA is '. Input::get() );
Log::info('User completed point x');
Log::notice('functionx is called');
Log::emergency('User was able to reach pageA without authentication');
```

You can use the preceding code at various points in your application to see what is happening in the application as users are testing it. You can see the output of your logging methods in `app/storage/logs/log-<date>.txt`. If you open your log file, you can see the output for each request will look like the following code snippet:

```
[2013-09-20 11:47:31] log.DEBUG: input data at pointA is, array( x=>1,
y=>2 )
[2013-09-20 11:47:31] log.INFO: User completed point x
[2013-09-20 11:47:31] log.NOTICE: functionx is called
[2013-09-20 11:47:31] log.emergency: User was able to reach pageA
without authentication. array( x=>1, y=>2 )
```

You will now have information to create a case for errors or users' complaints by asking just date and time to the user. You can find out what happened at that time and understand things much better; this should help you optimize your application.

Security in Laravel

One important aspect of managing web applications is security. There is a new security threat looming every few days. Let's see how Laravel helps secure our application and what things we would need to add to our application to make it secure.

Laravel's default authentication provides AES-256 encryption to passwords via a 32-bit key generated during the installation of Laravel in `app/config/app.php`. We can also use these algorithms to encrypt things in our application. In other words, if you want to encrypt confidential data, you can encrypt it as follows:

```
$encrypted = Crypt::encrypt('secret');
```

If you want to decrypt it, you can use the `decrypt` function as follows:

```
$decrypted = Crypt::decrypt($encryptedValue);
```

Similarly, Laravel has a secure hash with cookies, so if the client changes any of the cookies set by Laravel, Laravel will stop the request and redirect the user to an error page.

SQL injections

Laravel automatically handles SQL injection vulnerabilities for us via its `DB` class. It handles parameters by filtering them via prepared statements and uses **PDO (PHP Data Objects)**. So, as a user you, wouldn't have to worry about security issues.



Remember that if you are using `DB::raw`, which is used for sending raw expressions to databases, it will not be filtered by Laravel; therefore, be careful when you are using `DB::raw`.

CSRF

CSRF (Cross site request forgery) is a way to exploit websites using unauthorized commands that are transmitted from a third-party website accessed by a registered user while they are logged in to the site being targeted.

Laravel provides CSRF protection when you use the `Form::open` method to open forms. It adds a hidden token that will be checked if you have enabled the CSRF filter for your route. Here is how you can write your route:

```
Route::post('subscribe', array('before' => 'csrf', function()
{
    return 'subscribed';
}));
```

If you are not using the `Form::open` method to handle your forms, you can still add `csrf_token` via the following method:

```
<input type="hidden" name="_token" value="php echo csrf_token();
?&gt;"&gt;</pre

```

The preceding code will generate a CSRF token and send it when the form is posted.

XSS (Cross site scripting)

This is the one aspect that Laravel reserves for developers to decide on. XSS is always the most typical attack and is most used by hackers to steal information in the form of stealing a user's cookies and logging in via those cookies. The vulnerability exploited in this type of attack is that you need to allow your users to save content in HTML format, which is not encrypted. Hackers use the flaws in browsers or PHP or your code to bypass some sort of JavaScript code which will be executed and steal the user's cookies. So, the first rule you need to remember is, always filter the user's content.

If you do not intend to use any of your input for saving content in the HTML format, use the `strip_tags` PHP function to remove any tags from content saved by users. For example:

```
$name = strip_tags(Input::get('name'));
$address = strip_tags(Input::get('address'));
```

This is the most basic way of protecting your application from an XSS attack. But there are times when you need to allow users to save their content in HTML format. In these cases, use the following code to save the content:

```
function encode($field) {
    return htmlspecialchars($field, ENT_QUOTES, 'UTF-8');
}
$name = encode(Input::get('name'));
```

The preceding code will convert any special characters in user content into HTML entities. So any of the code which contains `<` (less than) becomes `<` and `>` (greater than) becomes `>`.

Any code that can be executed via JavaScript is not executed when echoed back. When you echo it, you will need to use the function `html_entity_decode` to display the content in properly formatted HTML.

Another important thing you need to remember is that your page should have the following meta tag within your HTML template:

```
<meta charset="UTF-8">
```

This is to make sure we are not using any other type of encoding that hackers can use to exploit security vulnerabilities.

We have learned how to manage our application's errors, optimize our application via profiler, and secure applications in Laravel.

Summary

In this chapter we have learned how to debug, profile, and secure our web applications. We first learned how to debug errors and profile our applications. We then learned how to log events. Finally, we learned how to secure our applications from SQL injections, CSRF, and XSS.

In the next chapter we will learn how to deploy web applications developed in Laravel.

10

Deploying Laravel Applications

In this chapter, we will learn how to deploy Laravel applications. We will see different methods that you can use to deploy Laravel applications. Also, we will see the configuration changes that you might need to make on some of the popular web hosts in order to work your application on their environment.

Imagine you have finished all your features in a project, now you want to make it live. How can you do that? To deploy any Laravel application, you will need to perform following steps:

- Creating production configuration
- Creating a directory structure based on your web host
- Uploading your Laravel application directory files
- Creating a database in the production site and upload your local database on the production site
- Giving proper permissions to your storage files
- Setting up `.htaccess` based on your server

Creating production configuration

Based on your web host, you may need to change your environment information such as the mail or cache provider, as well as the database information. If you change your local configuration files then your local application will stop working, and assuming that you are using Git, you may sometimes swipe the configuration files by mistake. To avoid all these problems you can use Laravel environments. You can define which environment will use which file. Head over to your `bootstrap/start.php` directory and change the following configuration:

```
$env = $app->detectEnvironment(array(

    'local' => array('your-machine-name'),
    'production' => array('example.com'),
));
```

Here, we have added the `production` entry in our `$env` array. This tells Laravel to look for the `production` folder in the `config` folder to override any configuration entries loaded from the `config` directory when requests are made from `example.com`.

So now say for example, you want to add your database information of `production` into `database.php`, then just create a new `database.php` file in the `production` directory in the `app/config` directory of your installation, and add following code to your `database.php`:

```
return array(
    'connections' => array(
        'mysql' => array(
            'driver' => 'mysql',
            'host' => 'db.example.com',
            'database' => 'myapp',
            'username' => 'myappuser',
            'password' => 'myapppassword',
            'charset' => 'utf8',
            'collation' => 'utf8_unicode_ci',
            'prefix' => '',
        ),
    ),
);
```

Now as our configuration is set up in the `bootstrap` file, Laravel will load the database connection from the `production` directory's `database.php`. Laravel will load other configurations from default files automatically. It will just override the ones you write in your configuration file.

So this is how you can have all your production settings as well as local settings in one place and without any conflicts.

Creating a directory structure based on your web host

There are times when you want to change the directory structure of Laravel before uploading it on the server, just as you have multiple sites on a server and you don't want to change the document root of your server to the Laravel's public directory as it will mean all other sites' roots will also change. Or when your shared server-hosting provider doesn't allow you to change the document root of your site. Then your best bet is to change the directory structure of your Laravel application before deploying it to meet the server's directory structure.

In order to do that, you would need to assume that your `public_html` or `www` folder from the web host is the public directory of your Laravel installation. So your public folder is uploaded to your `public_html` directory. What about other files? Well, you can create a directory named `Laravel` and put all files and directories there except the public directory which you will upload on the `public_html` directory. Now change the `index.php` file of your public directory to match the new path; that is, it would be `../Laravel/<existingpath>` and upload your `Laravel` directory in the parent directory of the `public_html` folder.

Uploading your Laravel application directory files

How do you upload your Laravel application files? Well, it depends on which type of web host you have and whether or not it provides you with the SSH access. Most of the shared hosts don't provide you with the SSH access or you have to file a ticket and verify yourself for that. Ask your web host support team if they provide SSH or not.

So there are two ways you can upload your Laravel application and they are:

- Via SSH
- Via FTP

Deploying via SSH

Secure Shell (SSH) is a protocol to secure data communication. If your web host permits SSH, then it will have an SSH server installed on the server. The SSH server allows you to open remote terminal connections which allow us to transfer files or run commands on the server. To connect with the SSH server, you will need to have an SSH client. SSH requires a client to use the SSH protocol to connect with the server and allow you to run commands.

There are different SSH clients available based on your operating system.

If you are on Windows operating system, you would need to use the following clients:

- **Putty:** <http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html>
- **Cygwin:** <http://www.cygwin.com/install.html>

If you are on Linux or Mac operating system, you don't need to install anything as both OS come with the OpenSSH client which allows us to use SSH.

Uploading files via OpenSSH (Linux, Mac)

Let's see how we can connect to a server via the OpenSSH client. In OpenSSH, you can connect with the remote server via the following command:

```
$ssh username@servername -p port
```

So for example, your site name is `example.com` and your user account is James, then you can log in to your server via following command:

```
$ssh james@example.com
```

Some shared web hosts don't allow the use of username and password to connect with your server. They restrict the SSH access based on SSH key files. Basically it works like how you would set up your SSH key via your web host admin panel (for example, **cPanel**). Via its key generator, you will generate a private key. The generator will give you a private SSH key, which can be matched with the public SSH key used by the server to authenticate your SSH session requests. This way there is no password involved and the security is maintained between two endpoints. Remember, you need to save your SSH keys in the `.ssh` directory located at `/home/user/`.

Now copy your Laravel directory, remove the vendors' directory, and compress the copied directory and use the following command to send your compressed directory to your web server:

```
$scp myapp.tar james@example.com:/home/var/www/myapp.tar
```

The preceding command will send `myapp.tar` from your system to the user's `www` directory which is an `apache` directory. Run the `composer` to install it via SSHs in the `application` directory of your server.

Uploading files via Putty (Windows)

If you are using Windows, then you would need to use the Putty SSH client for Windows. It doesn't operate the same way as the OpenSSH client. It has a graphical interface where you would need to fill the `hostname` (`example.com`), connection type (`ssh`), and then click on **Open** and it will load the command window connected with the SSH server and prompt you for your password.

Once you are logged in SSH to transfer your files, you would need to use the `pscp.exe` file which will be available on the Putty site from where you downloaded the Putty client. Once you have `pscp.exe` where you have Putty, you can then run the following command from the Putty window:

```
$pscp myapp.tar james@example.com:/var/www/
```

The preceding command will transfer your files from your directory to your web server's `root` directory.

Creating a database in the production site and uploading your local database on the production site

To upload your database, you would need to create a database in your web server. For shared hosts, there will be database options in your web panel (`cPanel`) where you can create your database and database user. For **Virtual Private Server (VPS)** or dedicated servers, you can create a new database and upload it by either installing **phpMyAdmin** or **Adminer** or directly **MySQL** command line if you are good with **MySQL** commands.

Now if you have the command line access to your server, you can use **Laravel's artisan** to import your database via the following command:

```
$ php artisan migrate
```

If you don't have access to command line, you will need to upload the database from your local copy of the database. Go to your **phpMyAdmin** or **Adminer** URL address and click on your database. Then, click on **import** and choose the database file you have from your local database. Once you click on **Upload**, your database is ready.

Giving proper permissions to your storage files

We also need to set correct permissions for storage files in the server. So, we need to give a write permission in the storage directory as follows:

```
$ chmod -R 777 /var/laravel/app/storage
```

The preceding command will change the storage directories permission to 777 recursively, so all files and directories will be writable for Laravel. Also, you will need to set a permission for any of your image upload folders in the public directory in the same way.

If your host doesn't allow permissions via command line, then you can use your web admin panel's file manager to change permissions.

Setting up .htaccess based on your server

Laravel does come with default .htaccess in your public folder which removes index.php from the URLs but different server configurations have different values. So if your server doesn't work with the default .htaccess, here is the .htaccess configuration you can try:

```
Options +FollowSymLinks
RewriteEngine On

RewriteCond %{REQUEST_FILENAME} !-d
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^ index.php [L]
```

If you are using **hostgator** as your server, then you would need to add the following line in your .htaccess file:

```
AddType application/x-httpd-php53 .php
```

This allows you to run your site in PHP 5.3, as by default sites will run in the older version of PHP as Laravel requires minimum PHP 5.3 Version. Similarly, if you are using Site5 as your web hosting provider, you need to add the following line to your .htaccess file:

```
AddHandler application/x-httpd-php53 .php
```

So check for the correct version of your PHP if your Laravel application is not working when you deploy it to live. You will find similar instruction at your web host support documents which will have exact syntax to enable PHP 5.3+ Versions.

Deploying via FTP

If you want to deploy via **File Transfer Protocol (FTP)**, you can do that simply by using the FTP client. Your web host will provide you with the hostname, username, and password which you can use to connect with your web server via FTP clients such as **FileZilla**. Once you are connected with your FTP program, you can simply transfer files from your local application directory to your remote directory such as `www` or `public_html`.

But the previous methods have a huge problem when you want to update the site. You would have to manually remember all file names and upload them by going to the particular folders where the file resides. There is a chance that you will forget one of the files one day and it may have a strange effect on your web application. The best way to deploy Laravel and update consistently in production is using Git.

Deploying via SSH from the Git repository

Git is a version management system and as you change your files it manages its version. The best part about it is that we can use Git to deploy the changed files from our local Git repository to the remote Git repository and Git automatically manages that process when we push our code to the remote Git repository.

You might be wondering how do we get our files from a Git repository? Well you can add post-receive hook which will run every time the Git repository is updated. So here is how you can do this.

First, you will need to have an SSH access in your server. Second, check if you have Git installed in your server. If it is not installed, then install it. Now create a directory under your `/var` directory.

```
$ sudo mkdir /var/git
```

Give permission to the apache user `www` to access the folder.

```
$ sudo chown -R :www /var/git
```

```
$ sudo chmod +s -R /var/git
```

Now we need to create a remote repository which will be updated by our local Git repository. To maintain the integrity of repository, we will make it a bare repository; that is, only the Git folders will be maintained. Files will not be shown in the Git directory.

```
$ mkdir /var/git/example.git
$ cd /var/git/example.git
$ git init --bare
```

Now we need to add a post-receive hook which will pick the change in our bare repository we just created and put the files in the site's `/var/www` folder.

So let's just add the post-receive hook by first creating a hook.

```
$ vim /var/git/appname/hooks/post-receive
```

And then add the following code to hook:

```
#!/bin/sh
GIT_WORK_TREE=/var/www/ git checkout -f
```

Also we need to make a Git hook executable so it can be executed via the following command:

```
$ chmod +x /var/git/example.git/hooks/post-receive
```

Now at our local Git repository, we need to add this remote repository via the following command:

```
$ git remote add myserver ssh://James@example.com:/var/git/appname.git
```

Now whenever you want to update your code in the production environment, run the following code:

```
$ git push myserver master
```

Deploying via FTP from the Git repository

If you have only the FTP access and you want to use Git to update your site, then you have to use a deployment package. One such package is **Dandelion**. To install Dandelion, you will need to have Ruby and RubyGems installed in your local system. Once you have installed Ruby and RubyGems, you can run the following command to get Dandelion:

```
$ gem install dandelion
```

Once you have Dandelion installed, you can deploy your updated Git files via the following command:

```
$ dandelion deploy
```

Hey, but where do we add our FTP username and password? Well, Dandelion searches the root of your Git repository for the configuration file named `dandelion.yml` which will contain the configuration to deploy the files to FTP.

So create `dandelion.yml` and update it with following values:

```
scheme: ftp
host: example.com
username: testuser
password: testpassword

# Optional
# -----

# Remote path
path: /var/www/

exclude:
  - .gitignore
  - dandelion.yml
  - tests/

# These files (from your working directory) will be uploaded on every
# deploy
additional:
  - public/css/print.css
  - public/css/screen.css
  - public/js/main.js
```

As you can see, it's a very simple configuration file with following options:

- **Scheme:** It would be either FTP or SFTP or Amazon S3
- **Path:** The path variable is used to denote where files will be uploaded in the production environment when you run Dandelion to deploy it
- **Exclude:** These files or folders will not be uploaded when you run the deploy command
- **Additional:** These files or folders will be uploaded on each deploy command

So that's how simple it is to use Dandelion to upload your files on a server via FTP.

Summary

In this chapter, we have learned how to deploy your web applications. We have seen two different techniques SSH and FTP to deploy your applications. We also looked at how Git could be used to automatically deploy applications via SSH and FTP. Also we learned the configuration needed to deploy things such as database and production related configurations.

In the next chapter, we will learn how to create workflow with some awesome Laravel packages and tools.

11

Creating a Workflow and Useful Laravel Packages and Tools

This chapter will describe some of the useful packages that can be used to create web applications rapidly in Laravel. This chapter will also give you some tips on tools and workflow which will make your life easy as a developer.

I think every developer spends almost an entire day writing code on their code editor, for the most part of their life. As developers, we should know editors better and learn tips and tricks to save our time, such as installing the add-ons, plugins, or packages it provides. For example, **Notepad++** provides a **Compare** plugin, which allows you to compare two files and shows the differences between those files. Now this feature might not be great unless you have two files – one, the backup file and the other, the current file. If you have added code that has stopped working and needs to be fixed, you can use this plugin to see what you actually changed in the code. (I know Git provides this, but at times, we have to deal with a file that doesn't have Git.) That's the reason you should know your editor better; those little features make your life easy and the time spent on learning the editor will help you through life, saving time and making your tasks easier every time.

As web developers, there are some aspects that are crucial in our day-to-day lives. The following are a few aspects:

- Debugging an application
- Command-line tools
- Version management via Git or SVN
- Testing tools
- Reusing code, components, or packages

Creating a workflow

From all the aspects described in the previous section, it would be great if the web developer's editor could do even a few from the editor window itself. So, that way, the developer doesn't have to constantly switch between windows and run tasks from other applications.

As a developer, I prefer to use the **Sublime** text editor as it manages almost four of the aspects we have previously mentioned. I have used a lot of editors until now and the list includes editors such as Notepad++, **Dreamweaver**, **Komodo IDE**, and **NetBeans**, all powerful editors with a lot of features and functionalities. If you are using them – great – and if you are still using Notepad or other editors, I would suggest using Sublime or any of the editors previously listed.

Now, in the scope of this chapter, I am going to cover the Sublime text editor, as I find that it's the only editor that can manage so many of the development aspects that I need to use in daily life.

I like the Sublime text editor because the first thing I noticed was that it's fast! Superfast, compared to other editors I have used. Looking for files is just *Ctrl + P* away (*command + P* on Mac). It lists all files via its fuzzy search and it's been almost accurate every time I have used it. It really helps when you are developing applications for which you don't want to use your mouse or need to open a sidebar tree, such as view, to find the file.

Another important aspect of the Sublime editor is Package Control. **Package Control** is the Sublime Text 2 package manager. It allows you to add the package's features. To install it, you need to run the python code on the console window of the Sublime editor. Let's install Package Control.

Visit <https://sublime.wbond.net/installation#st2> and copy the Sublime Text 2 code displayed on the tab window. Now, in your editor, click on the **view** menu and select the **Show Console** option, which will open the console window in Sublime Text 2. Just paste the code copied from the site, press *Enter*, and restart the editor. To see whether or not Package Control is installed, press *Ctrl + Shift+ P* (*command + Shift + P* on Mac) and input `install package`. If you can see the name appear while you are typing, you have Package Control installed.

Package manager allows us to install packages created by the developer community. We will see some really good packages that provide the development aspects that we discussed in the first section of the chapter.

Now, to debug PHP applications within our editor, we can install the **XDebug client** Sublime Text 2 package. To install this package, press *Ctrl + Shift + P* (*command + Shift + P* on Mac) and execute `install package`. This will fetch Sublime packages and display them in a list format; now search for the XDebug client and install it.

To use the XDebug client, we need to have XDebug installed in our installed server. If you are using XAMPP or WAMP in your local development environment, you might have the the XDebug extension installed within your application. To check whether or not XDebug is installed, you can run the `phpinfo()` function with the test PHP page and it will display all the extensions installed in the web server and check if its installed. If it's not installed, the guide to install it can be found at <http://xdebug.org/docs/install>.

You need to configure the XDebug package configuration file to debug the application. I would recommend using the Sublime project setting file to set the configuration as it gives you control to configure settings based on the project, which is useful as you will have different URLs for each project. Now create a `projectname.sublime` project file at the root of your folder and insert the following code:

```
{
  "folders":
  [
    {
      "path": "D:/projects/exampleproject"
    }
  ],
  "settings":
  {
    "XDebug": {
      "path_mapping": {

      },
      "ide_key": "sublime.XDebug",
      "url": "http://projectname.local",
      "port": 9000,
      "close_on_stop": true,
      "max_depth": 3,
      "max_children": 32,
      "debug": true
    }
  }
}
```


You need to input the URL of the local project that you want to debug in the `projectname.sublime` file, and the absolute path of your local directory in the `folders` array path configuration.

Now check your `php.ini` file and make sure `XDebug.remote_enable` is set to `on` or `1`. Then add a breakpoint within your code file from where you would like to debug via `Ctrl + Shift + F9`. Now open your project in the browser with the following URL:

```
http://projectname.local?XDEBUG_SESSION_START=1
```

Now if you switch back to your editor, you will see the debug window opened at the bottom with all the page variables and you can debug your code using the following shortcuts:

- Run: `Ctrl + Shift + F5`
- Step Over: `Ctrl + Shift + F6`
- Step Into: `Ctrl + Shift + F7`
- Step Out: `Ctrl + Shift + F8`
- Stop
- Detach

To stop the debugging process, press `Ctrl + Shift + P` and type `XDebug`; it will display the option to stop debugging or detach debugging. So this is how Sublime Text 2 helps you debug your application. The first aspect in our list is covered by Sublime Text 2. Let's see how other packages can help us.

Another package I really like is Git. This sublime package allows us to run Git commands in our editor, so we don't have to switch between our command window and editor. As web developers, we constantly have to work with the browser and editor, so adding the command window to that list is not a great option; so, we can use the Sublime Git to manage Git commands from the Sublime console.

To install the Sublime Git package, open the Sublime command window via `Ctrl + Shift + P` and select Git from the list. Restart the editor and you will have the Git package installed. To verify it, open the command window via `Ctrl + Shift + P` and type `git`; you will have all the Git commands arranged in a handy list (no need to remember them) and pretty good options to view the logfiles too.

Now say, for example, you want to create a Git repository; just open the command palette via `Ctrl + Shift + P`, type `git init`, and select the option. Your Git repository will be initialized. Generally, to do this you would have to open the command window and change the path to your project directory, then type `git init`. Compared to our workflow, it's less work; as you can see, you just have to type the `git init` command two times and you are done—no need to switch windows.

Another useful application of the Git package is when you are in an individual file and you want to see how that file looked in a specific commit. In such cases, you can type in `git log all` and it will display all the commits in a list. You can then select the commit number and it will display what it looked like and what has been changed between the current commit and the selected commit. Isn't that awesome?

So we have covered another aspect of our Git development. There are tons of packages for Sublime Text 2 that you can install. To search for packages, go to <https://sublime.wbond.net/> and view information about plugins. You can find plugins for adding snippets, changing the color scheme, or highlighting syntax for blade view files. I would recommend that you check out the following packages:

- **Emmet:** This helps you develop the HTML code faster via snippets.
- **phpcs:** This is the PHP Linter; that is, it checks for PHP syntax errors. It also adds functionalities such as coding the standard fixer and mess detector.
- **Livereload:** This reloads the project tab in the browser when you save the file, that is, there's no need to refresh the browser manually to test the application in it.

So we have learned a lot about the Sublime text editor and how we can use it to manage our workflow. Let's see some of the packages that really help us generate code via Laravel's artisan command-line tool.

Introducing JeffreyWay/Laravel-4-Generators

The first package I would like to introduce is **Laravel-4-Generators**. This package is really helpful when you are starting your application and want to build a prototype rapidly. It generates all types of CRUD (CREATE, READ, UPDATE, DELETE) boilerplate code, which we would otherwise have to write.

Let's install this package and see its usage in detail. To install the package, open up your `composer.json` file in the editor and add the following lines in the `require` array:

```
"require": {  
    "laravel/framework": "4.0.*",  
    "way/generators": "dev-master"  
},  
  
"minimum-stability" : "dev"
```

Run the `composer update` command so that Composer will download the package and update its files in the `vendor` directory. To integrate the package with Laravel, open up the `app.php` file from the `config` folder and add the service provider as follows:

```
'Way\Generators\GeneratorsServiceProvider'
```

Once you have added the service provider, the package will have added `generate` commands in the Laravel `artisan` commands. To see those commands, run `php artisan` and you will be able to see the new generator commands. Let's learn some of the important generators.

We can generate migration with fields via the migration generator provided in the generator package. So in just one line, you can generate your database tables using fields. So say, for example, you want to create an authors table with name, age, and ISBN number fields. You can do that by running the following command from the root of your project:

```
$ php artisan generate:migration create_authors_table
--fields="name:string, city:string, isbn_number: string"
$ php artisan migrate
```

The preceding command will create the authors table migration file in `app/database/migrations/` and then run the `migrate` command so the authors table will be created. Think about the convenience to this. You don't have to open your database editor, select the database, and add your fields. You just have to run one command and your database table is ready; that's the power of Laravel's `artisan` and generator package.

You can generate different types of files with structure via the generator package. Let's generate a model for our authors table using the following command:

```
$ php artisan generate:model Author
```

The preceding command will generate a model file in `app/model/author.php` with the following code:

```
<?php

class Author extends Eloquent {

}
```

You can use the model anywhere in your code; you don't need to change anything. Isn't that great? One command, and we don't have to write boilerplate code or create files and worry about structure. Similarly, just as we generated a model, you can generate the Controller, view, and database seeds.

Another generator feature is scaffolding. You can generate a CRUD code for your table without writing a single line of code. To generate a scaffold for our author's table, we can use the following command:

```
$ php artisan generate:scaffold Author --fields="name:string,  
age:integer, city:string, isbn_number:string"
```

The preceding command will generate a model, a resourceful Controller, view files, the migration file, the seed file, and the route declaration of our author's resource. All the code is actually generated within the files, so if you browse your application by going to `http://test.local/authors`, you will be able to access the interface that lists all the authors and, via that, the interface on which you can add/edit/delete authors. You can use this as a prototype when building an application rapidly.

Another awesome generator command is the forms generator. Forms generator accepts the model name as an argument and generates a form based on your fields. So, for example, if we need to generate our authors table, the command is:

```
$ php artisan generate:form Author
```

The preceding command will generate the following form:

```
{{ Form::open(array('route' => 'author.store')) }}  
    <ul>  
        <li>  
            {{ Form::label('name', 'Name:') }}  
            {{ Form::text('name') }}  
        </li>  
  
        <li>  
            {{ Form::label('age', 'Age:') }}  
            {{ Form::text('age') }}  
        </li>  
  
        <li>  
            {{ Form::label('city', 'City:') }}  
            {{ Form::text('city') }}  
        </li>  
  
        <li>  
            {{ Form::label('isbn_number', 'Isbn Number:') }}  
            {{ Form::text('isbn_number') }}
```

```
        {{ Form::text('isbn_number') }}
    </li>

    <li>
        {{ Form::submit() }}
    </li>
</ul>
{{ Form::close() }}
```

You can use this form in your view files directly.

Just like the generators, there are many other packages that you can use to quickly build prototypes. The following are some of the awesome packages that you can use to quick start your project:

- **Ardent:** Self-validating models for Laravel 4 (<https://github.com/laravelbook/ardent>)
- **Intervention:** The image manipulation package for Laravel 4 (http://intervention.olivervogel.net/image/getting_started/laravel)
- **Laravel 4 - Starter Kit:** More of a rapid prototype application that uses different Laravel packages with prebuilt features such as backend, frontend, and authentication (<https://github.com/brunogaspar/laravel4-starter-kit>)

Summary

In this chapter, we have learned how the workflow can save our development time. We learned how to utilize editor and Laravel packages to save time.

So, we have come a long way. Hope you have enjoyed the journey!

Index

Symbols

`$db` variable 85
`$errors` object 159
`$fillable` variable 48
`$guarded` variable 48
`$order_detail` object 170
`$user` object 185
`$user->save` method 185
`$users` object 182
`$validation` instance 48
`/app` directory 21
`/bootstrap` directory 21
`/commands` directory 21
`/config` directory 21
`/controllers` directory 21
`/database` directory 22
`.htaccess`
 setting up, based on server 220-223
`/lang` directory 22
`/models` directory 22
`/public` directory 21
`/start` directory 22
`/storage` directory 22
`--table` parameter 57
`/tests` directory 22
`/vendor` directory 21
`/views` directory 22

A

About Us page 72-74
`addgroup` method 156
`add` method 96
Add User button 180

`adduser` method 183
`Adminer` 219
administration section
 foundation, building 154-157
administrator
 login section, creating for 158-161
API method
 building, for individual store viewing 195
 creating, for store client deletion 199-201
 creating, for stores searching 196
Application class 89
application configurations
 debug 27
 key 28
 locale 28
 providers 28
 time zone 28
 URL 27
applications
 configuring, in Laravel 4 27, 28
Ardent 232
artisan command 60, 74, 90
artisan directory 21
Artisan tool
 about 28
 boilerplate controller, generating 29
 database, managing with migrations 29
 data, feeding with database seeds 29
 maintenance mode 30
 unit tests, running 29
authenticate method 121, 130

B

BaseController class 35
before parameter 194

boilerplate controller

generating 29

boot method 93

BuildCar function 86

C

Car class 84-88

CarserviceProvider class 88

cart

contents, viewing 104

deleting from 103

Foldagram, adding to 98-101

Foldagram, deleting from 117, 118

items, deleting from 104-106

total, viewing 104

updating 101-103

Cart class

creating 96-98

Cart::insert(\$array) method 111

Cart::insert method 112

Cart package

Foldagram order details,

adding to 111, 112

integrating, in Foldagram order process
106-108

changepassword method 135

check method 161

checkout method 142

checkout order process

creating 142, 143

checkout page

building, for credit cards 138-141

credits section, building 143-148

integrating 136-151

view orders section, building 149-151

checkout process 136

Compare plugin 225

Composer

about 11

benefit 11

installing 16

composer.json directory 21

Composer package archive

URL 11

Composer Update command 83, 230

Config::get method 112

contact method 35

Container class 86, 89

container variable 97

Controllers

about 34

versus, routes 35

Create method 48, 110

createUser method 127

credit

adding, for user 177-179

credits section

building 143-148

Cross site request forgery. *See* **CSRF**

Cross site scripting. *See* **XSS**

CRUD application

creating, Laravel 4 used 36, 37

new users, creating 45-49

user information, deleting 52

user information, editing 49-51

users, listing 37-45

users list pagination, adding 52

CSRF

about 211

web applications, securing from 211

Curl tool 199

D

Dandelion

installing 222

used, for file uploading 222, 223

data

logging, with Laravel 210

database

configuring, in Laravel 4 26, 27

filling, database seeds used 29

managing, migrations used 29

DatabaseSeeder class 59

database seeds

used, for database filling 29

DB class 27, 211

DB::table method 196

delete method 52, 199

Dependency Injection

about 84, 85

working 86-88

destroy() method 52, 199

- directory structure**
 - creating, based on web host 217
- div container** 70
- down function** 58

E

- edit method** 49
- Eloquent object** 27
- Eloquent ORM** 12
- Emmet** 229
- errors**
 - handling, with Laravel 205-207
- extends method** 129

F

- facades**
 - about 93, 94
 - creating 95
- fails method** 127
- FIG (Framework Interoperability Group)** 11
- File Transfer Protocol.** *See* **FTP**
- FileZilla** 221
- Filters.php directory** 22
- find method** 49, 51, 117, 199
- finduserbyid method** 185
- FinduserbyLogin method** 179
- Foldagram**
 - about 55
 - adding, to cart 98-101
 - deleting, from cart 117, 118
 - information, editing 116, 117
 - previewing 113-115
 - recipient information, deleting from 116
 - URL 55
- Foldagram form**
 - creating 77-81
- Foldagram information**
 - adding, to Foldagram table 106-108
- Foldagram order details**
 - adding, to Cart package 111, 112
- Foldagram order process**
 - Cart package, integrating in 106-108
- Foldagram preview** 78
- Foldagram pricing**
 - managing 175-177

- Foldagram table**
 - Foldagram information, adding to 106-108
- forget() method** 105
- Form class** 160
- Form::open method** 211
- Frameworks**
 - about 7
 - Laravel 4 Framework 7
- frontend**
 - creating, via REST API 201-204
- FTP** 221

G

- getFacadeAccessor() method** 93
- getIndex method** 180
- get method** 196
- getUser method** 134, 135
- git init command** 228
- Git repository**
 - Laravel application files, deploying
 - via FTP 222, 223
 - Laravel application files, deploying
 - via SSH 221, 222
- group routing** 157

H

- hidden property** 200
- Home page layout**
 - content section, creating 66, 67
 - Flexslider slider, setting up 67, 68
 - setting up 62
 - setting up, via Blade 62-64
- hostgator** 220
- html_entity_decode function** 212

I

- id parameter** 195
- Illuminate\View\view.php class** 94
- index method** 41, 53, 196
- initializecart method** 97
- inner pages**
 - setting up 72-74
- Input::all() function** 47
- Input::move method** 110

Input::old method 126
insert method 100, 103
Intervention 232
Intervention/image package
 used, for Laravel image resizing 109, 110
Inversion of Control. *See* **IoC container**
IoC container
 about 84
 working 87, 88
IoC container bindings
 handling, service providers used 88, 89

J

juv profiler 209

L

Laravel 4
 about 32
 Artisan command-line tool 28
 Controllers, creating in 35
 Controllers versus routes 35
 data, logging with 210
 errors, handling with 205-207
 image, resizing in 109
 installing 16, 17
 maintenance mode 30
 packages, creating in 89
 package structure 90-93
 REST API, creating 191, 192
 used, for CRUD application creating 36-52
 using 32-34
 working 89
Laravel 4 Framework
 about 7-10
 Composer tool 10-14
 configuring 23, 24
 features 11
 installing, on Linux (Ubuntu) 17, 18
 installing, on Mac 19-23
 installing, on Windows 16, 17
Laravel 4 Framework features
 Artisan 13
 auto loading 12
 Composer Ready 11

Eloquent ORM 12
events 13
interoperability 12
IOC containers 12
Migrations 13
pagination 13
Queues 13
RESTful 12
Routes 12
unit testing 13

Laravel-4-Generators

 about 229
 installing 229-232

Laravel 4 - Starter Kit 232

Laravel application files

 deploying, via FTP 221
 deploying, via FTP from
 Git repository 222, 223
 deploying, via SSH from
 Git repository 221, 222
 uploading 217
 uploading, via OpenSSH(Linux, Mac) 218
 uploading, via Putty (Windows) 219
 uploading, via SSH 218

Laravel applications

 deploying 215-223
 profiling 207-209

Laravel environment

 application, configuring 27, 28
 configuring 24-26
 database, configuring 26, 27

Laravel.log event 13

Laravel.query event 13

Laravel security

 about 210
 CSRF 211
 SQL injection 211
 XSS 212

layout

 setting up 61-72

link_to_route function 49

link_to_route() method 64, 66

Linux (Ubuntu)

 Laravel 4, installing on 17, 18

listIndex function 9

Livereload 229

M

Mac

- Laravel 4, installing on 19
- Laravel 4 structure, exploring 20-23

Mail class 127

make method 94

md5 method 99

migrate command 230

Migrations 57

migrations tool

- used, for database managing 29

modal function 114

Model class 48

mongodb class 85

msg variable 197

MVC

- about 8, 33
- code division 8
- web framework 9, 10

MVC code division

- Controller 8
- Models 8
- Views 8

myaccount method 131

mysql clas 85

MySQL command line 219

N

newsletter section

- creating, in Foldagram 74-77

nl2br function 115

Notepad++ 225

O

Object Rlational Mapper Package (ORM Package) 76

OpenSSH (Linux, Mac)

- Laravel application files, uploading via 218

order details section

- building 168-170

order management

- order details section, building 168-170
- orders, deleting 173, 174
- order status, updating 170-173
- view recipients section, building 167, 168

orders

- deleting 173, 174
- exporting 174
- managing 162-167

order status

- updating 170-173

P

Package Control 226

paginate method 52, 180

pagination class 180

paginator class 164, 167

passes() method 48

password

- changing 134
- updating 135
- validating 135

PATCH method 51

payment gateway

- integrating 136

payment options

- Pay via credit card 136
- Pay via credits 136

PDO (PHP Data Objects) 211

Pencil

- URL 56

phpcs 229

phpinfo() function 227

PHPMailer component 12

phpMyAdmin 219

Phpunit.xml file 91

postadd method 177

postedituser method 185

post_login method 129

postlogin method 160, 161

POSTMAN 199

POST method 51, 75, 160, 183

post_register method 126

postusercredit method 178

preview page

- creating, to preview Foldagram 113-115

production configuration

- creating 216

production site

- database, creating in 219
- local database, uploading on 219

- provides method 93
- Public directory 91
- Purchase Credit form section 147
- put method 101
- Putty (Windows)
 - Laravel application files, uploading via 219

Q

- query method 164

R

- recipient information
 - deleting, from Foldagram 116
- Recipients table
 - recipient information, adding to 110
- register method 88, 93, 123
- remove method 118
- removeRecipient method 116
- Representational State Transfer. *See* REST
- require attribute 120
- Resource Controllers
 - used, for REST API creating 191, 192
- REST
 - about 190
 - architecture 190
- REST API
 - creating, Resource Controllers
 - used 191, 192
 - creating, to view stores 195
 - frontend, creating via 201-204
 - store method, adding to 197
 - store method, updating 198
 - testing, curl used 199, 200
 - testing, POSTMAN used 200, 201
- REST architecture components
 - Clients 190
 - Resource 190
 - Servers 190
- RESTFUL backend
 - creating 192-194
- route method 123
- routes
 - versus, Controllers 35
- Routes.php directory 22
- routing 33

S

- save method 117
- saveUser method 38
- schema
 - Foldagram form, creating 77-80
 - inner pages, setting up 72-74
 - layout, setting up 61-72
 - newsletter section, creating 74-77
 - preparing 57-60
- Schema::create method 58
- Secure Shell. *See* SSH
- Seeder class 59
- select method 178
- send button 200
- Sentry
 - about 120, 154, 207
 - installing 120
 - Sentry::check() method 123, 157
 - Sentry::createUser method 156
 - Sentry::findGroupByName method 156
 - Sentry::findthrottlerByUserId method 186
 - Sentry::getUser method 157
- Sentry package
 - about 121
 - installing 120
- Sentry::Save method 185
- server.php directory 21
- ServiceProvider class 88
- service providers
 - used, for IoC container bindings
 - handling 88, 89
- session class 96, 97
- setcallback() method 195
- showUserRegistration method 38
- SQL injections
 - web applications, securing from 211
- src/ acme/ cart/ cartserviceprovider.php
 - directory 91
- Src/Config directory 91
- Src directory 91
- Src/Migrations directory 91
- Src/views directory 91
- SSH
 - about 218
 - Laravel application files, uploading via 218

- storage files**
 - proper permissions, providing to 220
- Store::find method 198**
- store locator**
 - single page web application, building 191-204
- storelocator method 203**
- store method 198**
 - adding, to API 197
 - updating 198
- Store() method 48**
- stores**
 - viewing, REST API used 195
- stores table**
 - about 192
 - fields 192
- Stripe payment gateway**
 - integrating 141
- strip_tags function 212**
- Sublime text editor**
 - about 226
 - advantages 226
 - Package Control 226
 - PHP applications, debugging 227
- Subscribe Eloquent object 76**
- suspend method 186**
- Symfony framework 83**

T

- Tests directory 91**
- third-party packages 21**
- timestamps() method 58**

U

- unit tests**
 - running 29
- update_cart() method 100, 103**
- update method 51, 103, 134, 198**
- up function 58**
- URL::to method 182**
- user**
 - adding 182-184
 - blocking 186
 - credit, adding for 177-179
 - deleting 185
 - editing 185

- managing 179-182
 - registering 123-126
 - validating 127

- User class 42**

- UserController class 38**

- user dashboard page**

- Controller tab, creating 131-133
 - creating 130, 134

- UserExistsException exception 157**

- user login**

- setting up 128-130

- User object 156**

- UserController class 51, 53**

- user section**

- dashboard page, creating 130-134
 - login, setting up 128-130
 - password, changing 134, 135
 - registration screen, creating 123-127
 - setting up 121-123

- users table 59**

- about 192
 - fields 193
 - fields, adding 194

V

- validate method 99**

- Validation class 47, 127**

- validation object 76**

- Validator class 130, 134**

- Vendor directory 91**

- View class 94**

- view object 94**

- view orders section**

- building 149-151

- view recipients section**

- building 167, 168

- Virtual Private Server (VPS) 219**

W

- web application**

- building, of store locator 191

- web host based directory structure**

- creating 217

- Windows**

- Composer, installing on 16

- Laravel 4, installing on 16, 17

workbench command 24

workflow

creating 226-229

World Wide Web Consortium (W3C) 190

X

XDebug client

configuring 227

used, for PHP applications debugging 227

XSS

about 212

web applications, securing from 212



Thank you for buying **Learning Laravel 4 Application Development**

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

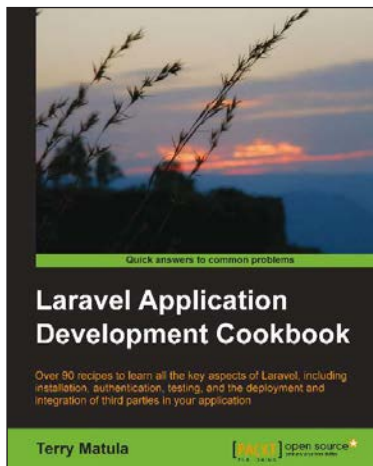
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



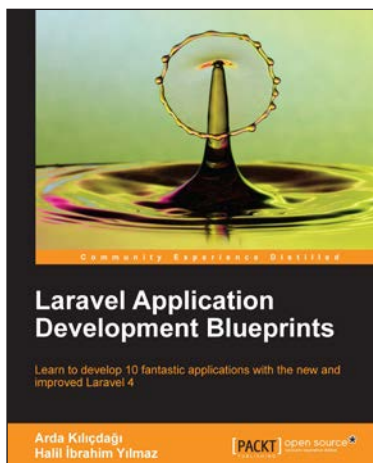
Laravel Application Development Cookbook

ISBN: 978-1-78216-282-7

Paperback: 272 pages

Over 90 recipes to learn all the key aspects of Laravel, including installation, authentication, testing, and the deployment and integration of third parties in your application

1. Install and set up a Laravel application and then deploy and integrate third parties in your application
2. Create a secure authentication system and build a RESTful API
3. Build your own Composer Package and incorporate JavaScript and AJAX methods into Laravel



Laravel Application Development Blueprints

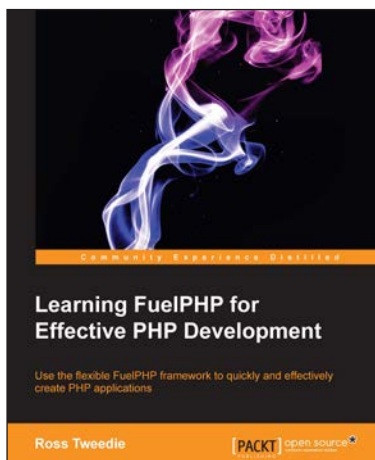
ISBN: 978-1-78328-211-1

Paperback: 260 pages

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

1. Learn how to integrate third-party scripts and libraries into your application
2. With different techniques, learn how to adapt different methods to your needs
3. Expand your knowledge of Laravel 4 so you can tailor the sample solutions to your requirements

Please check www.PacktPub.com for information on our titles



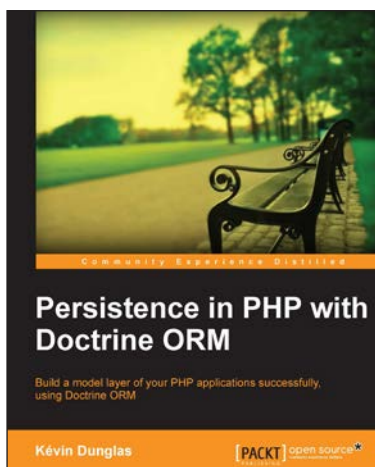
Learning FuelPHP for Effective PHP Development

ISBN: 978-1-78216-036-6

Paperback: 104 pages

Use the flexible FuelPHP framework to quickly and effectively create PHP applications

1. Scaffold with oil - the FuelPHP command-line tool
2. Build an administration quickly and effectively
3. Create your own project using FuelPHP



Persistence in PHP with Doctrine ORM

ISBN: 978-1-78216-410-4

Paperback: 104 pages

Build a model layer of your PHP applications successfully, using Doctrine ORM

1. Develop a fully functional Doctrine-backed web application
2. Demonstrate aspects of Doctrine using code samples
3. Generate a database schema from your PHP classes

Please check www.PacktPub.com for information on our titles