

Community Experience Distilled

Mastering Laravel

Develop robust modern web-based software applications and RESTful APIs with Laravel, one of the hottest PHP frameworks

Christopher John Pecoraro

[PACKT] PUBLISHING open source*

Mastering Laravel

Table of Contents

Mastering Laravel
Credits
About the Author
About the Reviewers
www.PacktPub.com
    Support files, eBooks, discount offers, and more
        Why subscribe?
        Free access for Packt account holders
Preface
    What this book covers
    What you need for this book
    Who this book is for
    Conventions
    Reader feedback
    Customer support
        Downloading the example code
        Errata
        Piracy
        Questions
1. Designing Done Right with phpspec
    A new era
        A leaner app
        PSR
    Installing and configuring Laravel
        Installation
        Configuration
        Namespacing
    TDD done right
        PHPUnit
    phpspec
    Entity creation
    The MyCompany database schema
    Designing with phpspec
    Specifying with phpspec
        Red, green, refactor

- Tidying things up
- Controllers
  - The command bus
- Summary

2. Automating Tests – Migrating and Seeding Your Database
- Using Laravel's migration feature
  - An example of migration
  - Creating the table
  - The Laravel migration magic
  - $table->timestamps();
- From schema to migration
  - Composer's require-dev command
  - Laravel's providers array
  - The composer update command
  - Generating the migrations
- Migration anatomy
  - List tables
  - The softDelete and timestamp properties
- Creating seeds
- Database testing with PHPUnit
  - Running PHPUnit
- Functional testing with Behat
- Summary

3. Building Services, Commands, and Events
- Request routing
  - User stories
  - User stories to code
  - The controller
    - Searching for the room
    - Controller to command
    - Command to event
    - The ReserveRoomCommandHandler class
    - Event to handler
  - Queued event handlers
  - The waiting list command
  - The queued commands
  - The console command
  - The command scheduler
- Summary

4. Creating RESTful APIs
    RESTful APIs in Laravel
    Essential CRUD
    Bonus features
    Controller creation
    CRUD(L) by example
        cRudl – read
        crudL – list
            Pagination
        Crudl – create
        crUdl – update
        cruDl – delete
    Model binding
        Read revisited
        List revisited
        Update revisited
        Delete revisited
    Moving beyond CRUD
    Nested controllers
        Accommodation hasMany rooms
        Room belongsTo accommodation
        Eloquent relations
        Nested update
        Nested create
    Eloquent model casting
    Route caching
    Summary
5. Using the Form Builder
    History
    Installing the HTML package
    Building web pages with Laravel
        The master template
        An example page
        From static HTML to static methods
            The form tag
            The text input field
            The label tag
            Checkbox
            The submit button

The anchor tag with links
Closing the form
The resultant form
Our example
Month select
Date select
Year select
Form macros
Conclusion
Summary
6. Taming Complexity with Annotations
Annotations in other programming languages
Annotations in Java
Annotations in C#
Annotations in PHP
DocBlock annotations
DocBlock annotations in Laravel
Symfony
Zend
Laravel
Resource controller using DocBlock annotations
Single method routing
Scanning routes
Automatic scanning
Additional annotations
HTTP verbs
Other annotations
Using annotations in Laravel 5
Advantages
Conclusion
Summary
7. Filtering Requests with Middleware
The HTTP kernel
The basic middleware structure
Route middleware unravelled
Default middleware – the Authenticate class
Contracts
Handle
Custom middleware – Log

The Log model
Log model migration
Terminable middleware
Logging as terminable
Using middleware
Route groups
Multiple middleware in route groups
Middleware exclusion and inclusion
Conclusion
Summary
8. Querying the Database with the Eloquent ORM
Basic operations
Finding one
The where method
Chaining functions
Finding all
Eloquent relations
One-to-one
One-to-many
Many-to-many
Has-many-through
Polymorphic relations
Amenitiable relationships
The Amenity table structure
The Amenity model
The Accommodation model
The Room model
Many-to-many polymorphic relations
Has relationships
Eager loading
Conclusion
Summary
9. Scaling Laravel
Scalability issues
Towards the enterprise
Route caching
Illuminate routing
Lumen
Comparison between Laravel and Lumen

Lean application development
Read/write
Master table
Slave table
Configuring read/write
Creating a master/slave database configuration
Master server set up
Slave server set up
Summary
10. Building, Compiling, and Testing with Elixir
Automating Laravel
Deployment
Development or deployment
Towards automation
From Gulp to Elixir
Getting started
Installing Node.js
Installing the Node.js package manager
Installing Gulp
Installing Elixir
Running Elixir
Setting up notifications
Combining CSS and JavaScript files with Elixir
Compiling with Laravel Elixir
Compiling Sass and Less
Compiling CoffeeScript
A summary of compiler commands
Putting everything together
Running tests with Elixir
PHPSpec
PHPUnit
Creating custom tasks
Setting up a file watcher
Additional Laravel Elixir tasks
Summary
Index

Mastering Laravel

Mastering Laravel

Copyright © 2015 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: July 2015

Production reference: 1270715

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78528-502-8

www.packtpub.com

Credits

Author

Christopher John Pecoraro

Reviewers

Kevin Coyle

Js Lim

Chinmoy Maity

Commissioning Editor

Sarah Crofton

Acquisition Editor

Tushar Gupta

Content Development Editor

Sumeet Sawant

Technical Editors

Ankur Ghiye

Saurabh Malhotra

Narsimha Pai

Copy Editors

Roshni Banerjee

Trishya Hajare

Vedangi Narvekar

Rashmi Sawant

Project Coordinator

Shweta H Birwatkar

Proofreader

Safis Editing

Indexer

Hemangini Bari

Graphics

Sheetal Aute

Production Coordinator

Nitesh Thakur

Cover Work

Nitesh Thakur

About the Author

Christopher John Pecoraro is an experienced web application developer born near Pittsburgh, Pennsylvania. After earning his BS. degree in computing and information science at Saint Vincent College in 1999, the majority of his career has centered around web application development in both the United States and Europe.

He is a conference speaker, open source contributor, and an author. His research work includes biomedical informatics and machine translation, and he is a coauthor of several peer-reviewed publications. His native language is English and he speaks fluent Italian; he has visited many countries, and his noncareer interests are travel and cuisine.

About the Reviewers

Kevin Coyle has been developing web applications for nearly 10 years. With a keen interest in all technological things, he has become a polyglot for various programming languages.

In the past, he has led the Drupal/PHP development in one of the largest telecommunications companies in the UK and has run his own small development agency specializing in people in the media.

I would like to thank my family for putting up with me through the late nights while I was reviewing this book and my work colleagues for advising me to put the time in to do the review. Of course, most of all, I would like to thank my pet beagle, Archer, who, without fail, never ceases to make me smile.

Js Lim had totally no idea about programming back in 2008. He started to pick up programming when doing an assignment at Tunku Abdul Rahman college, Malaysia, using the C programming language. Then, in 2011, he entered into a company as an internship trainee. This was the first time he studied PHP. His company was using two frameworks: one was an in-house framework, which was quite messy, and the other one was CodeIgniter. At that moment, the in-house framework needed to be refactored; thus, he and his team managed to code the AdoDB ORM plugin into that and changed it to an MVC design, which took quite some time. He is very fresh and is always looking for new things to learn. He saw his supervisor using Ubuntu and Vim, and he followed that the rest of the staff was using Windows and Notepad++. He is now a Vim lover.

After his graduation, he joined a company that was into developing enterprise projects using ColdFusion. After working there for 3 months, he left due to the technologies used. He joined a company named Werebits, which was a start-up. This company was doing an in-house project, ChopInk, which is a kind of loyalty program. The technology behind this was LAMP stack-hosted on Amazon EC2, Zend Framework 1. After a few months, the company took a project, which he chose to use Zend Framework 2 on, and took some time to finish the project. Later on, the CEO decided to migrate ChopInk to another framework. He was assigned the task of choosing a better framework; after some research, he decided to use Laravel 4. Up until now, Laravel is still the best framework for him as well as to the Werebits team. At some point, he was approached by a college friend to join a

company named Weavary Connection as a partner; he then got the technology over, which was the Laravel framework.

Chinmoy Maity is a programmer, entrepreneur, and loves to write code for a living.

He graduated in 2010 with a BSc in computer science with a first-class grade. He completed has MCA (master of computer application) degree in 2013 with a first-class grade and was awarded a gold medal.

He has been working in the web development industry for the last two and a half years (mostly with start-ups).

Since 2007, he has been coding, and this has become his profession—he is a backend developer. He is always excited to learn new technologies, write new code, and learn new programming languages. He is an expert in building core PHP and PHP framework-based applications. Other than being good at frontend (such as HTML, CSS, JavaScript, jQuery, AJAX, responsive designing, and so on), he does development too. Besides all his professional work, he works on a few hobby projects that are related to Android and robotics.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

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.



https://www2.packtpub.com/books/subscription/packtlib

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read 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 a 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 9 entirely free books. Simply use your login credentials for immediate access.

This book is dedicated to my mother for giving me life, my father for giving me wisdom, and my wife for her patience and love.

Dogni mirada tua est una prenda, virtude chi possedis naturale

Preface

PHP, a free and open source programming language, is continuing renaissance and Laravel is at the forefront. Laravel 5 is proving to be the most usable framework for novice and expert programmers alike. Following modern PHP's object-oriented best practices, reduce the time to market and build robust web and API-driven mobile applications that can be automatically tested and deployed.

You will learn how to rapidly develop software applications using the Laravel 5 PHP framework.

What this book covers

Chapter 1, *Designing Done Right with phpspec*, speaks about how to configure Laravel 5 for phpspec to perform modern unit testing, to use phpspec to design classes, and to perform unit and functional testing.

Chapter 2, *Automating Tests – Migrating and Seeding Your Database*, covers database migrations, the mechanics behind them and how to create a seed for testing.

Chapter 3, *Building Services, Commands, and Events*, talks about Model-View-Controller and how has evolved into services, commands, and events to decouple code and practice separation of concerns.

Chapter 4, *Creating RESTful APIs*, takes you through the creation of a RESTful API: the basic CRUD operations (create, read, update, and delete), as well as discussing some best practices and hypermedia controls (HATEOAS).

Chapter 5, *Using the Form Builder*, takes you to the web interface side of things and shows you how to take advantage of the some of the newest features of Laravel 5 to create web forms. Reversed routing will be discussed here as well.

Chapter 6, *Taming Complexity with Annotations*, focuses on annotations. The `routes.php` file easily becomes messy when an application grows in complexity. Using annotations inside of controllers, code legibility is drastically increased; however, there are some disadvantages in addition to the advantages.

Chapter 7, *Filtering Requests with Middleware*, shows you how to create reusable filters that can be called either before or after the controllers.

Chapter 8, *Querying the Database with the Eloquent ORM*, helps you learn how to use an ORM in a way to reduce the probability of errors in coding, increase the security and reduction of SQL-injection probability, and also learn how to deal with the limits of the Eloquent ORM.

Chapter 9, *Scaling Laravel*, speaks about scaling an application to move it into a cloud-based architecture. The read/write master/slave configuration is discussed and the reader is guided through the configuration.

Chapter 10, *Building, Compiling, and Testing with Elixir*, introduces Elixir. Elixir is based on gulp, which is a task runner and is a series of build scripts that automates common tasks in the Laravel software development workflow.

What you need for this book

We'll need the following software:

- Apache/Nginx
- PHP 5.4 or greater
- MySQL or similar
- Composer
- phpspec
- Node.js
- npm

Who this book is for

If you are an experienced novice or a capable PHP programmer who has a basic understanding of the concepts of modern PHP (at least version 5.4), then this book is ideal for you.

Basic object-oriented programming and database knowledge is expected. You should already know your way around Laravel or will have at least experimented with the framework.

Conventions

In this book, you will find a number of text styles 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: “ The new `artisan` command is run as follows”

A block of code is set as follows:

```
protected function schedule(Schedule $schedule)
    {
        $schedule->command('inspire')
             ->hourly();
        $schedule->command('manage:waitinglist')
            ->everyFiveMinutes();

    }
```

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

```
$ php artisan schedule:run
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: “ The **migration** table now appears, as shown in the following screenshot.”

Note

Warnings or important notes appear in a box like this.

Tip

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 disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail <feedback@packtpub.com>, and mention the book's title in 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 at 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 from your account at http://www.packtpub.com for all the Packt Publishing books you have purchased. 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 could 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 to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to https://www.packtpub.com/books/content/support and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted 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.

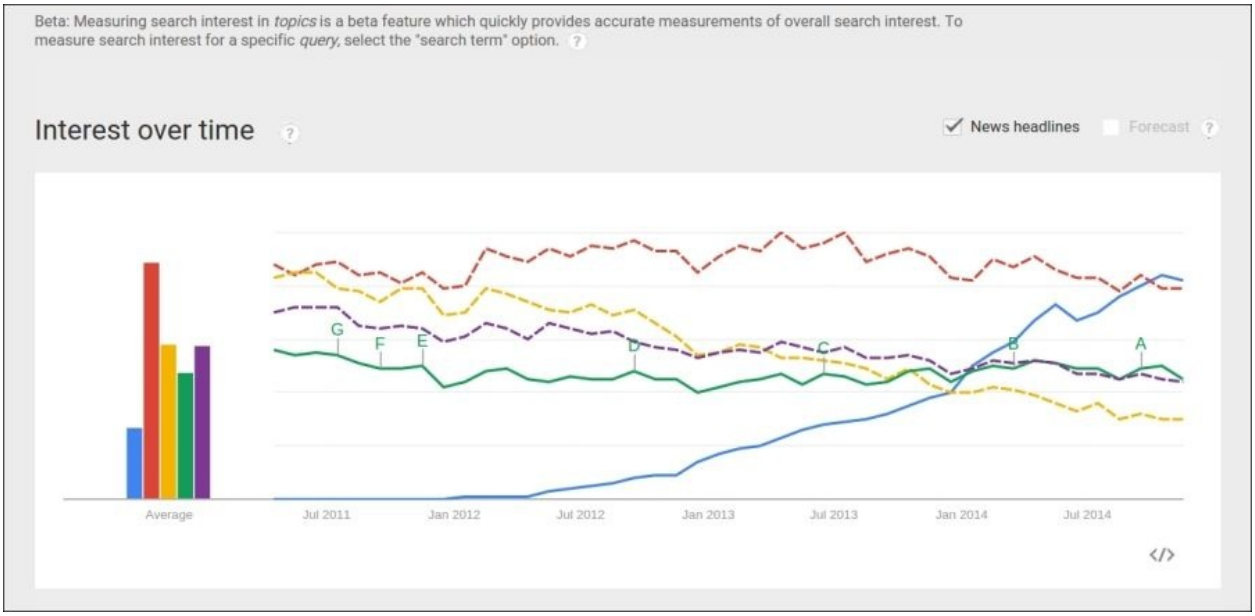
Questions

If you have a problem with any aspect of this book, you can contact us at <questions@packtpub.com>, and we will do our best to address the problem.

Chapter 1. Designing Done Right with phpspec

Many things have happened since Laravel's humble beginnings in 2011. Taylor Otwell, a .NET programmer, sought out PHP as a way to do a side project, since he was told that hosted PHP was cheap and ubiquitous. What originally started out as an extension to CodeIgniter become its own code. Freeing up the code base from the limitations of CodeIgniter's PHP 5.2, all of the new features that PHP 5.3 had to offer, such as namespacing and closures, could be used. The time span between the release of versions 1 and 3 of Laravel was only one year. With version 3, things happened very quickly. After its explosion in popularity, which happened around the time that version 4 was released, it quickly began to steal the market share from other popular frameworks such as CodeIgniter, Zend, Symfony, Yii, and CakePHP to eventually take the pole position. Along with its expressive syntax, great documentation, and a passionate founder came large community mainstays the IRC and Slack chat room, The Laravel Podcast, and the Laracasts instructional video website. Also, the newly created commercial support such as Envoyer, which provides *100 percent uptime*, means that Laravel was also welcomed by enterprises. With the release of Laravel 4.2, the minimum required PHP version was increased to 5.4 to take advantage of modern PHP features such as *traits*.

Using Laravel's traits along with the new syntax, such as the [] array shortcut, makes coding a breeze. Laravel's expressive syntax, coupled with these modern PHP features, makes it a great choice for any developer who wishes to build robust applications.



Laravel's rise in success, as reported by Google Trends

A new era

In late 2014, the second most important part of the history of Laravel occurred. When what was scheduled to be version 4.3 changed many of the core principals of Laravel, the community decided that it should become version 5.

The arrival of Laravel 5 brings about many changes in the way we use it to build software. The built-in MVC architecture that was inherited from frameworks such as CodeIgniter has been abandoned in favor of being more dynamic, modular, and even daringly framework-agnostic. Many of the components have been decoupled as much as possible. The most important part of Laravel's history will be the arrival of Laravel version 5.1, which will have **long-term support** (**LTS**). Thus, Laravel's place in enterprises will be solidified even more. Also, the minimum PHP requirements will be changed to version 5.5. So, for any new projects, PHP 5.5, or even PHP 5.6, is recommended because upgrading to PHP version 7 will be even easier.

A leaner app

The `/app` directory was slimmed down, leaving in only the most essential parts of the application. Directories such as `config`, `database`, `storage`, and `tests` have been moved out of the `app` directory since they are auxiliary to the application itself. Most importantly, the integration of the testing tools has matured drastically.

PSR

Thanks to the efforts of the **Framework Interoperability Group** (**PHP-FIG**), the developer of the **PHP Standard Recommendation** (**PSR**), the reading, writing, and formatting of the framework code is becoming easier. It even allows developers to more easily work in more than one framework. Laravel is a part of the FIG and continues to adopt its recommendations into the framework. Laravel 5.1, for example, will adopt the PSR-2 standard. For more information about the PHP FIG and PSR, visit the PHP-FIG website, http://www.php-fig.org.

Installing and configuring Laravel

The latest up-to-date instructions to install Laravel can always be found at the Laravel website, http://laravel.com. To begin using Laravel in a development environment, the current best practices suggest using the following:

- **Vagrant**: This provides a convenient way to manage a virtual machine, such as Virtualbox.
- **PuPHPet**: This is an excellent tool that can be used to create a virtual machine of various types. For more information about PuPHPet, visit https://puphpet.com.
- **Phansible**: This is an alternative to PuPHPet. For information about Phansible, visit http://phansible.com.
- **Homestead**: This is maintained by the Laravel community, and is a virtual machine that is created specifically for Laravel and which uses NGINX instead of Apache. For more information about Homestead, visit https://github.com/laravel/homestead.

Installation

The basic process involves downloading and installing Composer and then adding Laravel as a dependency. An important detail is that the storage directory, which is located parallel to the `/app` directory, needs to be set in such a way that it is writable by the web server user in order to allow Laravel 5 to do things like writing the log files. It is also important to make sure that `$ php artisan key:generate` is used to generate a 32-character key that is used for hashing because, since the release of PHP 5.6, Mcrypt is more strict as regards its requirements. For Laravel 5.1, OpenSSL will replace Mcrypt.

Configuration

In Laravel 4, the environments were configured in a manner that relied on the hostname of the server or a development machine, and this was rather contrived. Laravel 5 instead uses a `.env` file that sets up the various environments. This file is included in `.gitignore`. Thus, each machine should receive its configuration from a source outside the source code control.

So for example, something like the following code can be used to set up a local development:

```
APP_ENV=local
APP_DEBUG=true
APP_KEY=SomeRandomString
DB_HOST=localhost
DB_DATABASE=example
DB_USERNAME=DBUser
DB_PASSWORD=DBPass
CACHE_DRIVER=file
SESSION_DRIVER=file
```

Namespacing

A nice new feature of Laravel is that it allows you to set the highest level namespace to something such as `MyCompany` through the `app:name` command. This command will actually change the namespace of all the relevant files inside the `/app` directory from App to `MyCompany`, for example. This namespace then lives inside the `/app` directory. This builds namespacing right into virtually every file whereas previously, in version 4.x, it was optional.

TDD done right

The culture of test-driven development is not new. Rather, it has been around even before Kent Beck wrote SUnit in the 1990's. The xUNIT family of unit testing frameworks, which stemmed from SUnit, has grown to provide a testing solution for PHP.

PHPUnit

The PHP port of the PHP testing software is named PHPUnit. Yet, test-driven development in the PHP language is a fairly recent concept. For example, in his book, "*The Grumpy Programmer's Guide To Building Testable PHP Applications*", which was published at the end of 2012, *Chris Hartjes* wrote "I started looking into the culture of testing surrounding CodeIgniter. It's weaker than a newborn baby."

Testing has been a part of the Laravel framework since version 3 that uses the PHPUnit unit-testing tool, and therefore Laravel's inclusion of the `phpunit.xml` file was a huge leap forward in the effort to encourage developers to embrace test-driven development.

phpspec

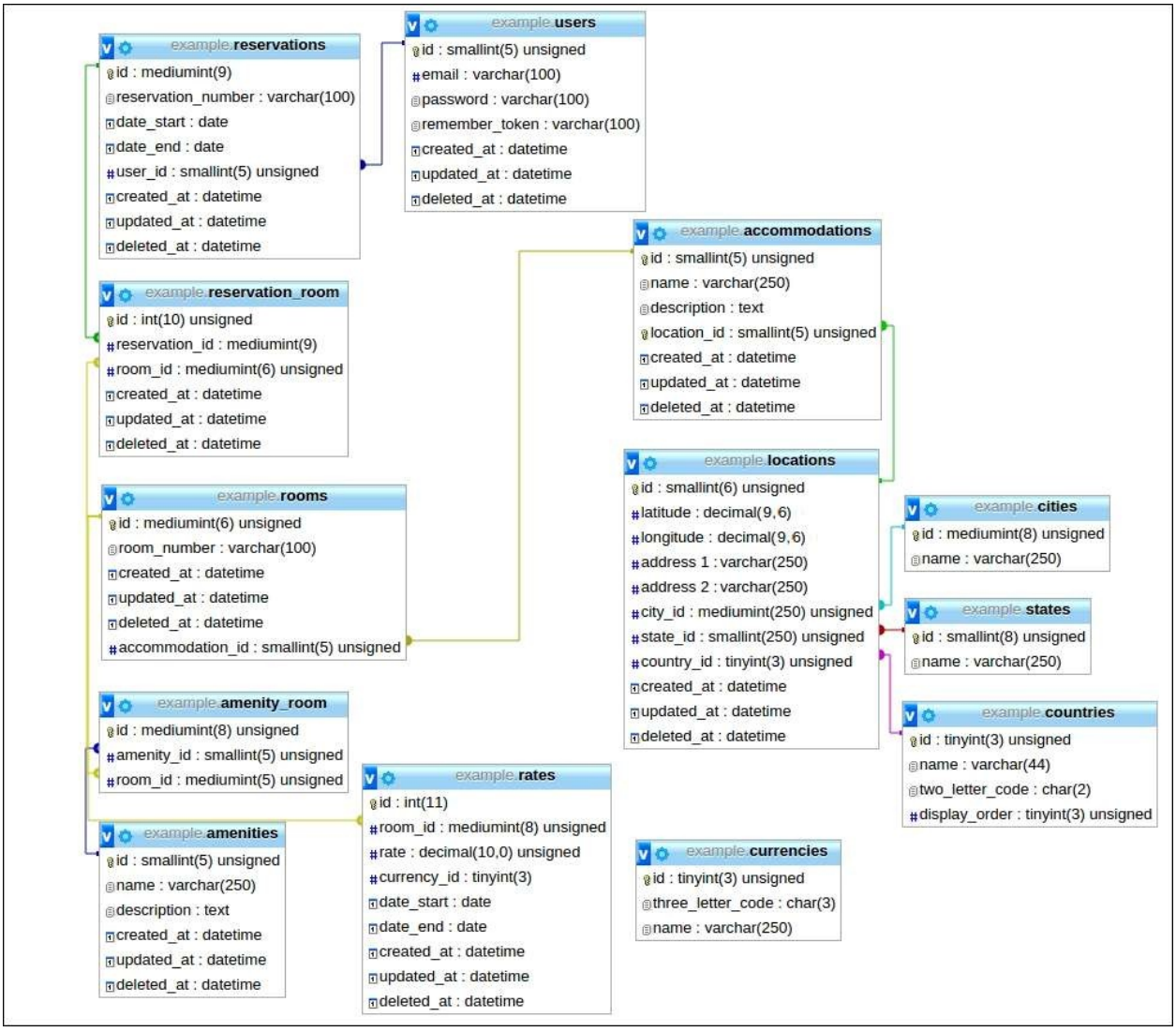
Another testing tool, RSpec, emerged in the Ruby community in 2007 and was a refinement on test-driven development. It featured **behavior driven development** (**BDD**). The phpspec tool, which brought ported RSpec's BDD to PHP, is growing rapidly in popularity. Its co-creator, Marcello Duarte, repeatedly states that "BDD is TDD done right". Therefore, BDD is simply an *improvement* or evolution of TDD. Laravel 5 now cleverly includes phpspec as a way to accentuate the *design by specification* paradigm of behavior-driven development.

Since an essential step in building a Laravel 5 application is to specify which entities to create, after installing and configuring Laravel 5, the developer may immediately start designing by running phpspec as a design tool.

Entity creation

Let's create a sample web application. If the client has asked us to build a booking system for tourism structures, then the system may contain the entities such as accommodations (hotels and bed and breakfasts, for example), rooms, rates, and reservations.

The simplified database schema will look like this:



The MyCompany database schema

The database schema has the following assumptions:

- An accommodation has many rooms
- Reservations are made for a single user
- A reservation may include more than one room
- A reservation has a start date and an end date
- Rates are valid for one room from a start date to an end date
- A room has many amenities
- The start date of the reservation must come before the end date
- A reservation cannot be made for more than fifteen days
- A reservation cannot include more than four rooms

Designing with phpspec

Now, let's begin using phpspec as a design tool to build our entities.

If the top-level namespace is `MyCompany`, then use phpspec and simply type the following command:

```
# phpspec describe MyCompany/AccommodationRepository
```

On typing the preceding command, `spec/AccommodationSpecRepository.php` gets created:

```
<?php

namespace spec\MyCompany;

use PhpSpec\ObjectBehavior;
use Prophecy\Argument;

class AccommodationRepositorySpec extends ObjectBehavior
{
    function it_is_initializable()
    {
        $this-
>shouldHaveType('MyCompany\AccommodationRepository');
    }
<?php

namespace MyCompany;

class AccommodationRepository
{
}
```

Tip

The path to phpspec should be added to either the `.bashrc` or the `.bash_profile` file so that phpspec can be run directly.

Then, type the following command:

```
# phpspec run
```

On typing the preceding command, the developer is shown, as follows:

```
class MyCompany\AcccommodationRepository does not exist.
Do you want me to create 'MyCompany\AccommodationRepository' for
you? [Y/n]
```

After typing *Y*, the `AccommodationRepository.php` class is created, as follows:

```
<?php

namespace MyCompany;

class AccommodationRepository
{}
```

Tip

Downloading the example code

You can download the example code files from your account at http://www.packtpub.com for all the Packt Publishing books you have purchased. 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 beauty of phpspec lies in its simplicity and the ability to speed up the creation of classes which carry along with the specification.



The basic steps involved in describing and creating a class with phpspec

Specifying with phpspec

At the core of phpspec is the ability to allow us to specify the behavior of entities and simultaneously test them. By simply specifying what the business rules are as given by the customer, we can easily create tests for each business rule. However, the real power of phpspec lies in how it uses an expressive, natural language syntax. Let's take a look at the business rules that were previously given to us regarding reservations:

- The start date of the reservation must come before the end date
- A reservation cannot be made for more than fifteen days
- A reservation cannot include more than four rooms

Run the following command:

```
# phpspec describe
 MyCompany/Accommodation/ReservationValidator
```

phpspec will produce the following output for the preceding command:

```
<?php

namespace spec\MyCompany\Accommodation;

use PhpSpec\ObjectBehavior;
use Prophecy\Argument;

class ReservationSpec extends ObjectBehavior
{
    function it_is_initializable()
    {
        $this-
>shouldHaveType('MyCompany\Accommodation\Reservation');
    }
}
```

Then, run phpspec by using the following command:

```
# phpspec run
```

phpspec will respond as usual with the following output:

```
Do you want me to create
  'MyCompany\Accommodation\ReservationValidator' for you?
```

Then, phpspec will create the `ReservationValidator` class, as follows:

```
<?php namespace MyCompany\Accommodation;

 class ReservationValidator {
 }
```

Let's create a `validate()` function that will take the following parameters:

- A start date string that determines the start of the reservation
- An end date string that determines the end of the reservation
- An array of `room` objects to add to the reservation

The following is the code snippet that creates the `validate()` function:

```
<?php
namespace MyCompany\Accommodation;

use Carbon\Carbon;

class ReservationValidator
{

    public function validate($start_date, $end_date, $rooms)
    {
    }
}
```

We will include the `Carbon` class, which will help us to work with the dates. For the first business rule, which states that the start date of the reservation must come before the end date, we can now create our first specification method in the `ReservationValidatorSpec` class, as follows:

```
function its_start_date_must_come_before_the_end_date
($start_date,$end_date,$room)
{
    $rooms = [$room];
    $start_date = '2015-06-03';
    $end_date = '2015-06-03';
    $this->shouldThrow('\InvalidArgumentException')-
>duringValidate( $start_date, $end_date, $rooms);
}
```

In the preceding function, phpspec starts the specification with `it` or `its`. phpspec uses the snake case for high legibility, and `start_date_must_be_less_than_the_end_date` is an exact copy of the

specification. Isn't this just wonderful?

When `$start_date`, `$end_date`, and the `room` are passed, they automatically get mocked. Nothing else is needed. We will create a `$rooms` array that is valid. However, we will set the `$start_date` and `$end_date` in such a way that they both have the same values to cause the test to fail. The expression syntax is shown in the preceding code. The `shouldThrow` comes before `during`, which then takes the method name `Validate`.

We have given phpspec what it needs to automatically create the `validate()` method for us. We will specify that `$this`, which is the `ReservationValidator` class, will throw an `InvalidArgumentException`. Run the following command:

```
# phpspec run
```

Once again, phpspec asks us the following:

```
 Do you want me to create
'MyCompany\Accommodation\Reservation::validate()'
  for you?
```

By simply typing *Y* at the prompt, the method is created inside the `ReservationValidator` class. It is that easy. When phpspec is run again, it will fail because the method has not thrown an exception yet. So now, the code needs to be written. Inside the function, we will create two `Carbon` objects from a string that is formatted like "2015-06-02" so that we are able to harness the power of Carbon's powerful date comparisons. In this case, we will use the `$date1->diffInDays($date2);` method to test whether the difference between the `$end` and the `$start` is less than one. If this is the case, we will throw the `InvalidArgumentException` and display a user-friendly message. Now, when we rerun phpspec, the test will pass:

```
$end = Carbon::createFromFormat('Y-m-d', $end_date);
$start = Carbon::createFromFormat('Y-m-d', $start_date);

        if ($end->diffInDays($start)<1) {
            throw new \InvalidArgumentException('Requires end
date to be greater than start date.');
        }
```

Red, green, refactor

The rules of test-driven development call for *red*, *green*, *refactor*, which means that once the tests pass (green), we should try to refactor or simplify the code inside the method without altering the functionality.

Have a look at the `if` test:

```
if ( $end->diffInDays($start) < 1 ) {
```

The preceding code isn't quite readable. We can refactor it in the following way:

```
if (!$end->diffInDays($start)>0)
```

However, even the preceding code is not very legible, and we are also using an integer directly in the code.

Let's move `0` into a constant. To improve the readability, we'll change it to the minimum amount of days required for a reservation, as follows:

```
 const MINIMUM_STAY_LENGTH = 1;
```

Let's extract the comparison into a method, as follows:

```
    /**
     * @param $end
     * @param $start
     * @return bool
     */
    private function endDateIsGreaterThanStartDate($end, $start)
    {
        return $end->diffInDays($start) >= MINIMUM_STAY_LENGTH;
    }
```

We can now write the `if` statement like this:

```
if (!$this->endDateIsGreaterThanStartDate($end, $start))
```

The preceding statement is much more expressive and readable.

Now, for the next rule, which states that a reservation cannot be made for more than fifteen days, we'll need to create the method in the following way:

```
function it_cannot_be_made_for_more_than_fifteen_days(User $user,
```

```
$start_date, $end_date, Room $room)
{
        $start_date = '2015-06-01';
        $end_date = '2015-07-30';
        $rooms = [$room];
        $this->shouldThrow('\InvalidArgumentException')
        ->duringCreateNew( $user,$start_date,$end_date,$rooms);
}
```

Here, we set the `$end_date` so that it is assigned a date that occurs more than a month after the `$start_date` to cause the method to throw an `InvalidArgumentException`. Once again, upon execution of the `phpspec` command, the test will fail. Let's modify the existing method to check the date range. We'll add the following code to the method:

```
  if ($end->diffInDays($start)>15) {
       throw new \InvalidArgumentException('Cannot reserve a room
       for more than fifteen (15) days.');
  }
```

Once again, phpspec happily runs all the tests successfully. Refactoring, we will once again extract the `if` condition and create the constant, as follows:

```
   const MAXIMUM_STAY_LENGTH = 15;
   /**
     * @param $end
     * @param $start
     * @return bool
     */
    private function daysAreGreaterThanMaximumAllowed($end,
$start)
    {
        return $end->diffInDays($start) >
self::MAXIMUM_STAY_LENGTH;
    }

   if ($this->daysAreGreaterThanMaximumAllowed($end, $start)) {
            throw new \InvalidArgumentException ('Cannot reserve
a room for more than fifteen (15) days.');
   }
```

Tidying things up

We could leave things like this, but let's clean it up since we have tests. Since the `endDateIsGreaterThanStartDate($end, $start)` and `daysAreGreaterThanMaximumAllowed($end, $start)` functions both check for the minimum and maximum allowed stay respectively, we can call them from another method.

We will refactor `endDateIsGreaterThanStartDate()` into `daysAreLessThanMinimumAllowed($end, $start)` and then create another method that checks both the minimum and maximum stay length, as follows:

```
private function daysAreWithinAcceptableRange($end, $start)
    {
        if ($this->daysAreLessThanMinimumAllowed($end, $start)
            || $this->daysAreGreaterThanMaximumAllowed($end,
$start)) {
           return false;
        } else {
           return true;
        }
    }
```

This leaves us with simply one function, instead of two, in the `createNew` function, as follows:

```
if (!$this->daysAreWithinAcceptableRange($end, $start)) {
            throw new \InvalidArgumentException('Requires a stay
length from '
                . self::MINIMUM_STAY_LENGTH . ' to '.
self::MAXIMUM_STAY_LENGTH . ' days.');
        }
```

For the third rule, which states that a reservation cannot contain more than four rooms, the process is the same. Create the specification, as follows:

```
it_cannot_contain_than_four_rooms
```

The change here will be in the parameters. This time, we will mock five rooms so that the test will fail, as follows:

```
function it_cannot_contain_than_four_rooms(User $user,
$start_date, $end_date, Room $room1, Room $room2, Room $room3,
Room $room4, Room $room5)
```

Five room objects will be loaded into the `$rooms` array, and the test will fail as follows:

```
$rooms = [$room1, $room2, $room3, $room4, $room5];
    $this->shouldThrow('\InvalidArgumentException')-
>duringCreateNew($user,$start_date,$end_date,$rooms);
    }
```

After adding code to check the size of the array, the final class will look like this:

```
<?php

namespace MyCompany\Accommodation;

use Carbon\Carbon;
class ReservationValidator
{

    const MINIMUM_STAY_LENGTH = 1;
    const MAXIMUM_STAY_LENGTH = 15;
    const MAXIMUM_ROOMS = 4;

    /**
     * @param $start_date
     * @param $end_date
     * @param $rooms
     * @return $this
     */
    public function validate($start_date, $end_date, $rooms)
    {
        $end = Carbon::createFromFormat('Y-m-d', $end_date);
        $start = Carbon::createFromFormat('Y-m-d', $start_date);

        if (!$this->daysAreWithinAcceptableRange($end, $start)) {
            throw new \InvalidArgumentException('Requires a stay
length from '
                . self::MINIMUM_STAY_LENGTH . ' to '.
self::MAXIMUM_STAY_LENGTH . ' days.');
        }
        if (!is_array($rooms)) {
            throw new \InvalidArgumentException('Requires last
parameter rooms to be an array.');
        }
        if ($this->tooManyRooms($rooms)) {
            throw new \InvalidArgumentException('Cannot reserve
more than '. self::MAXIMUM_ROOMS .' rooms.');
        }

        return $this;
```

```
    }

    /**
     * @param $end
     * @param $start
     * @return bool
     */
    private function daysAreLessThanMinimumAllowed($end, $start)
    {
        return $end->diffInDays($start) <
self::MINIMUM_STAY_LENGTH;
    }

    /**
     * @param $end
     * @param $start
     * @return bool
     */
    private function daysAreGreaterThanMaximumAllowed($end,
$start)
    {
        return $end->diffInDays($start) >
self::MAXIMUM_STAY_LENGTH;
    }

    /**
     * @param $end
     * @param $start
     * @return bool
     */
    private function daysAreWithinAcceptableRange($end, $start)
    {
        if ($this->daysAreLessThanMinimumAllowed($end, $start)
            || $this->daysAreGreaterThanMaximumAllowed($end,
$start)) {
            return false;
        } else {
            return true;
        }
    }

    /**
     * @param $rooms
     * @return bool
     */
    private function tooManyRooms($rooms)
    {
        return count($rooms) > self::MAXIMUM_ROOMS;
    }
```

```
    public function rooms(){
        return $this-
>belongsToMany('MyCompany\Accommodation\Room')->withTimestamps();
    }

}
```

The method is very clean. There are only two `if` statements—the first to verify that the date range is valid, and other one to verify that the number of rooms is within the valid range. The constants are easily accessible and can be changed as the business requirements change. Clearly, the addition of phpspec into the development workflow combines what earlier required two steps—writing the assertions with PHPUnit and then writing the code. Now, we will leave phpspec and move on to Artisan, which developers are familiar with as it was a feature of the previous versions of Laravel.

Controllers

Next, we'll create some sample controllers. At the time of writing this book, we need to use Artisan and phpspec together. Let's create a controller for the `room` entity, as follows:

```
$ php artisan make:controller RoomController

<?php namespace MyCompany\Http\Controllers;

use MyCompany\Http\Requests;
use MyCompany\Http\Controllers\Controller;

use Illuminate\Http\Request;
class RoomController extends Controller {

        /**
        * 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()
        {}
….

}
```

Note

Note that this will be created in `app/Http/Controllers` directory, which is a new location for Laravel 5. The new HTTP directory houses the controllers,

middleware, and requests directories, grouping together the files related to the HTTP request or the actual request. Additionally, this directory configuration is optional, and routes can call any autoloaded location, usually through a namespaced PSR-4 structure.

The command bus

Laravel 5 has adopted the command bus pattern, which creates commands that get created in the `app/Commands` directory. Whereas commands in Laravel 4 were thought of as command-line tools, in Laravel 5, the command is thought of as a class whose methods can be used from within the application, allowing for an excellent reuse of code. The concept of a command here is a task that needs to be done, or in our example, a room to be reserved for a user. The paradigm of a bus then transports the command using the new `DispatchesCommands` trait, which is used in the base controller class. Every controller created by Artisan extends this class to a handler method, where the actual work is performed.

To use Laravel's command bus design pattern, we'll now use Artisan to create a few commands. We'll go into detail about commands in a future chapter, but to begin, we will type the following command:

```
$ php artisan make:commandReserveRoomCommand --handler
```

Typing this creates a command to reserve a room which could be called from anywhere in the code, isolating the business logic from the controllers and models, and also allowing the command to be executed in an asynchronous mode.

```
<?php namespace MyCompany\Commands;

use MyCompany\Commands\Command;

class ReserveRoomCommand extends Command {

    /**
    * Create a new command instance.
    *
    * @return void
    */
    public function __construct()
    {
        //
    }

}
```

After filling in the details of the command, the class will now look like this:

```
<?php namespace MyCompany\Commands;
```

```
use MyCompany\Commands\Command;
use MyCompany\User;

class ReserveRoomCommand extends Command {

    public $user;
    public $rooms;
    public $start_date;
    public $end_date;

    /**
    * Create a new command instance.
    *
    * @return void
    */
    public function __construct(User $user, $start_date,
$end_date, $rooms)
    {
        $this->rooms = $rooms;
        $this->user = $user;
        $this->start_date = $start_date;
        $this->end_date = $end_date;
    }

}
```

The `--handler` parameter creates an additional class, `ReserveRoomCommandHandler`, containing a constructor and a handle method, which injects the `ReserveRoomCommand`. This file will be present in the `app/Handlers/Commands` directory. If the `--handler` flag is not used, then the `ReserveRoomCommand` class will contain its own `handler` method, and the separate handler class will not get created:

```
<?php namespace MyCompany\Handlers\Commands;

use MyCompany\Commands\ReserveRoomCommand;

use Illuminate\Queue\InteractsWithQueue;

class ReserveRoomCommandHandler {

    /**
    * Create the command handler.
    *
    * @return void
    */
    public function __construct()
    {
```

```
        //
    }

    /**
    * Handle the command.
    *
    * @paramReserveRoomCommand  $command
    * @return void
    */
    public function handle(ReserveRoomCommand $command)
    {
        //
    }

}
```

We will fill in the handle method with the validation of the reservation, as follows:

```
public function handle(ReserveRoomCommand $command)
    {
        $reservation = new
\MyCompany\Accommodation\ReservationValidator();
        $reservation->validate(
        $command->start_date, $command->end_date, $command-
>rooms);
    }
```

Summary

phpspec adds a mature, robust, test-first, test-driven, and a specification-by-example approach to creating the business logic aspect of the software. This, coupled with the ease of creation of models, controllers, commands, events, and event handlers, sets Laravel at the front of the PHP framework race. Also, it has adopted many best practices that are used by the best programmers in the industry.

In this chapter, we learned how to use phpspec to easily design classes and their accompanying tests from the command line. This workflow, accompanied by Artisan, makes the process of setting up the basic structure of a Laravel 5 application very easy.

In the next chapter, we'll take a look at database migrations, the mechanics behind them, and ways to create a seed for testing.

Chapter 2. Automating Tests – Migrating and Seeding Your Database

So far, we have created some base models and the general outline of the database. Now, we need to create database migrations and seeding. Traditionally, database "dump" files have been used as a way to pass around both the schema, which is the structure of the tables, and the data, which would be the initial or predefined records, such as default values; unchanging lists, such as cities or countries; and users such as "admin". These dump files that contain SQL can be committed to source code control. This is not always the best way to maintain the integrity of the database; since every time a developer adds records or modifies the database, all of the developers in the team would need to drop and recreate the database or add or delete the data, tables, rows, columns, or indexes manually. Migrations allow the database to live in the form of code, actually residing inside the Laravel project, as well as to be versioned within source code control.

Migrations are run from the command line and can also be automated to automatically create the database, whenever required if it doesn't already exist, or drop and recreate the tables and populate the tables if they already exist. Migrations have existed for a while in Laravel, so their presence in Laravel 5 is not surprising.

Using Laravel's migration feature

The first step is to run the `artisan` command:

```
$ php artisan migrate:install
```

This will create a table named `migration`, which has two columns: `migration`, which is a varchar 255 in MySQL, and `batch`, which is an integer. This table will be used by Laravel to keep track of which migrations have been run. In other words, it maintains a history of all of the operations that have been performed. The following is a list of the main operations:

- `install`: As mentioned earlier, this operation installs
- `refresh`: This operation resets and reruns all of the migrations
- `reset`: This operation rolls back all of migrations
- `rollback`: This operation is a type of "undo", and simply rolls back the last operation
- `status`: This operation produces a table-like output of the migrations and states whether or not they have been run

An example of migration

Laravel 5 contains two migrations in the `/database/migrations` directory.

The first migration creates the `users` table.

The second one creates the `password_resets` table, which, as you may have guessed, is used to recover lost passwords. Unless specified, the migrations operate on the database that is configured in the `/config/database.php` configuration file:

```
<?php

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

class CreateUsersTable extends Migration {

  /**
   * Run the migrations.
   *
   * @return void
   */
  public function up()
  {
    Schema::create('users', function(Blueprint $table)
    {
      $table->smallIncrements('id')->unsigned();
      $table->string('name');
      $table->string('email')->unique();
      $table->string('password', 60);
      $table->rememberToken();
      $table->timestamps();
      $table->softDeletes();
    });
  }

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

```
}
```

Migrations extend the `Migration` class and use the `Blueprint` class.

There are two methods: `up` and `down`, which are used when the `migrate` commands and the `rollback` commands are used, respectively. The `Schema::create()` method is called with the table name as the first parameter and a function callback as the second parameter, which accepts an instance of a `Blueprint` object as a parameter.

Creating the table

The `$table` object has a few methods that perform tasks such as creating indexes, setting up auto-increment fields, stating which type of fields should be created, and passing the name of the field as a parameter.

The first command is used to create an auto-increment field `id`, which will be the primary key of the table. Then, string fields, such as `name`, `email`, and `password` are created. Note that the unique method is chained to the `create` statement for the `email` field, stating that the `email` field will be used as the login name/user ID, as this is a common practice in most modern web applications. The `rememberToken` is used to allow the user to remain authenticated per each session. This token gets reset on each login and logout, protecting the user from a potentially malicious hijacking attempt.

The Laravel migration magic

Laravel migrations are also capable of creating timestamp fields that are used to automatically store creation and update information for each model through its table row.

$table->timestamps();

The following line of code tells the migration to automatically create two columns in the table, namely `created_at` and `updated_at`, which is automatically used by Laravel's Eloquent **Object-relational mapping** (**ORM**) to allow the application to know when the object was created and when it was updated:

```
$table->timestamps()
```

In the following example, the fields are updated as follows:

```
/*
*   created_at is set with timestamps
*/
$user = new User();
$user->email = "johndoe@acmewidgets.com";
$user->name = "John Doe";
$user->save(); // created_at is set with timestamps

/*
*   updated_at is set with timestamps
*/
$user = User::find(1); //where 1 is the $id
$user->email = "johndoe@acmeenterprise.com";
$user->save(); //updated_at is updated
```

Another great Laravel feature is the soft delete field. This provides a type of **recycle bin** to allow the data to be optionally restored at a later time.

This feature simply adds another column to the table to allow soft deleting of the data. The code to be added to the migration looks like this:

```
$table->softDeletes();
```

This adds a column to the `database, deleted_at,` which will either have `null` as its value or a timestamp to indicate when the record was deleted. This builds a recycle bin feature right into your database application.

Run the following command:

```
$ php artisan migrate
```

The migration is initiated and the tables are created. The **migration** table now appears, as shown in the following screenshot:

migration	batch
2014_10_12_000000_create_users_table	1
2014_10_12_100000_create_password_resets_table	1

The structure of the `users` table is shown in the following screenshot:

Name	Type	Collation	Attributes	Null	Default	Extra
id	int(10)		UNSIGNED	No	*None*	AUTO_INCREMENT
name	varchar(255)	utf8_unicode_ci		No	*None*	
email	varchar(255)	utf8_unicode_ci		No	*None*	
password	varchar(60)	utf8_unicode_ci		No	*None*	
remember_token	varchar(100)	utf8_unicode_ci		Yes	*NULL*	
created_at	timestamp			No	0000-00-00 00:00:00	
updated_at	timestamp			No	0000-00-00 00:00:00	
deleted_at	timestamp			Yes	*NULL*	

To rollback the migration, run the following command:

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

The `rollback` command uses the migration table to determine which actions to rollback. In this case, the `migrations` table, after it has been run, is now empty.

From schema to migration

A common situation that occurs during the development process is that a schema is created, and then, we need to create a migration from that schema. At the time of writing, there is no official tool to do this in the Laravel core, but there are several packages available.

One such package is the `migrations-generator` package.

First, add the following line to the `require-dev` section of the `composer.json` file to require the `migrations-generator` dependency in the `composer.json` file:

```
"require-dev": {
    "phpunit/phpunit": "~4.0",
    "phpspec/phpspec": "~2.1",
    "xethron/migrations-generator": "dev-feature/laravel-five-
stable",
    "way/generators": "dev-feature/laravel-five-stable"
  },
```

It is also necessary to add the following text to the `composer.json` file at the root level:

```
"repositories": [
  {
    "type": "git",
    "url": "git@github.com:jamisonvalenta/Laravel-4-
Generators.git"
  }],
```

Composer's require-dev command

The `require-dev` command, as opposed to `require`, is a mechanism of a composer that allows certain packages that are needed only in the development phase. Most testing tools and migration tools will only be used in the local development machine, QA machine, and/or in a continuous integration environment, but not in the production environment. This mechanism keeps your production installation free of unnecessary packages.

Laravel's providers array

Laravel's `providers` array in the `config/app.php` file lists the providers that are available to Laravel at all times.

We will add both the `way generator` and the `Xethron migration` service providers:

```
'providers' => [

        /*
         * Laravel Framework Service Providers...
         */

Illuminate\Foundation\Providers\ArtisanServiceProvider::class,
          Illuminate\Auth\AuthServiceProvider::class,

Illuminate\Broadcasting\BroadcastServiceProvider::class,
        ...
    'Way\Generators\GeneratorsServiceProvider',

'Xethron\MigrationsGenerator\MigrationsGeneratorServiceProvider'
]
```

The composer update command

The `composer update` command is a simple, yet powerful way to make sure that everything that needs to be in place is actually working and free of errors. After running this command, we're now ready to run the migrations.

Generating the migrations

Simply type the following command:

```
$ php artisan
```

The `artisan` command will display a list of all of the possible commands. The `migrate:generate` command should be included on the list of valid commands. If this command is not on the list, then something is not configured correctly.

Once you confirm that the `migrate:generate` command exists in the list, simply run the following command:

```
$ php artisan migrate:generate
```

This will start the process.

In this example, we have used the MySQL database. By entering `Y` when prompted, the process will begin and the output should show one migration file created for each table in the database.

This is how your command prompt should appear at the end:

```
Using connection: mysql

Generating migrations for: accommodations, amenities,
amenity_room, cities, countries, currencies, locations, rates,
reservation_room, reservations, rooms, states, users
Do you want to log these migrations in the migrations table?
[Y/n] Y
Migration table created successfully.
Next Batch Number is: 1. We recommend using Batch Number 0 so
that it becomes the "first" migration [Default: 0]
Setting up Tables and Index Migrations
Created:
/var/www/laravel.example/database/migrations/2015_02_07_170311_cr
eate_accommodations_table.php
Created:
/var/www/laravel.example/database/migrations/2015_02_07_170311_cr
eate_amenities_table.php
Created:
/var/www/laravel.example/database/migrations/2015_02_07_170311_cr
eate_amenity_room_table.php
Created:
/var/www/laravel.example/database/migrations/2015_02_07_170311_cr
```

```
eate_cities_table.php
Created:
/var/www/laravel.example/database/migrations/2015_02_07_170311_cr
eate_countries_table.php
Created:
/var/www/laravel.example/database/migrations/2015_02_07_170311_cr
eate_currencies_table.php
Created:
/var/www/laravel.example/database/migrations/2015_02_07_170311_cr
eate_locations_table.php
Created:
/var/www/laravel.example/database/migrations/2015_02_07_170311_cr
eate_rates_table.php
Created:
/var/www/laravel.example/database/migrations/2015_02_07_170311_cr
eate_reservation_room_table.php
Created:
/var/www/laravel.example/database/migrations/2015_02_07_170311_cr
eate_reservations_table.php
Created:
/var/www/laravel.example/database/migrations/2015_02_07_170311_cr
eate_rooms_table.php
Created:
/var/www/laravel.example/database/migrations/2015_02_07_170311_cr
eate_states_table.php
Created:
/var/www/laravel.example/database/migrations/2015_02_07_170311_cr
eate_users_table.php

Finished!
```

Migration anatomy

Consider an example of one of the lines in the migration file; we can see that the table object is used in a chain of methods. The following line of the migration file sets up the state attributes in the location eloquent attribute in the `locations` table:

```
$table->smallInteger('state_id')->unsigned()->index('state_id');
```

List tables

Often, it is necessary to create or import a list of finite items that usually remain constant, such as cities, states, countries, and similar items. Let's call these list tables or lookup tables. In these tables, the ID should usually be positive. These lists may grow, but they usually will not have any data that is deleted or updated. The `smallInteger` type is used to keep the table small and also represent a value that belongs to a finite list, something that will not grow naturally. The next method, `unsigned`, states that the limit will be 65535. This value should be enough to represent most of the states, provinces, or similar types of geographical regions where a hotel could be located. The last method in the chain adds an index to the database column. This is essential in list tables like these, which are used in the `select` statements or in the `read` statements. The `Read` statements will be discussed in Chapter 9, *Scaling Laravel*. It is important to use unsigned, as it doubles the positive limit, which otherwise would be 32767. Using the index, we can speed up the look up time and access a cached version of the data in the table.

The softDelete and timestamp properties

Regarding `softDeletes` and `timestamps` for list tables, it depends. If the table is not very large, it shouldn't be too harmful to keep track of any updates, inserts, or deletions if they do happen; however, if the list consists of countries, where changes occur infrequently and are very small, it would be prudent to omit `softDeletes` and `timestamps`. So, the entire table will probably fit into the memory and will be very fast. To omit timestamps, it's necessary to add the following line of code:

```
public $timestamps = false;
```

Creating seeds

To create our database seeder, we will modify the `DatabaseSeeder` class that extends `Seeder`. The name of the file is `database/seeds/DatabaseSeeder.php`. The contents of the file will be as follows:

```
<?php

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

class DatabaseSeeder extends Seeder {

    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        Model::unguard();

        //create a user
        $user = new \MyCompany\User();
        $user->id=1;
        $user->email = "testing@tester.com";
        $user->password = Hash::make('p@ssw0rd');
        $user->save();

        //create a country
        $country = new \MyCompany\Accommodation\Location\State;
        $country->name = "United States";
        $country->id = 236;
        $country->save();

        //create a state
        $state = new \MyCompany\Accommodation\Location\State;
        $state->name = "Pennsylvania";
        $state->id = 1;
        $state->save();

        //create a city
        $city = new \MyCompany\Accommodation\Location\City;
        $city->name = "Pittsburgh";
        $city->save();

        //create a location
```

```
        $location = new \MyCompany\Accommodation\Location;
        $location->city_id = $city->id;
        $location->state_id = $state->id;
        $location->country_id = 236;
        $location->latitude = 40.44;
        $location->longitude = 80;
        $location->code = '15212';
        $location->address_1 = "100 Main Street";
        $location->save();

        //create a new accommodation
        $accommodation = new \MyCompany\Accommodation;
        $accommodation->name = "Royal Plaza Hotel";
        $accommodation->location_id = $location;
        $accommodation->description = "A modern, 4-star hotel";
        $accommodation->save();

        //create a room
        $room1 = new \MyCompany\Accommodation\Room;
        $room1->room_number= 'A01';
        $room1->accommodation_id = $accommodation->id;
        $room1->save();

        //create another room
        $room2 = new \MyCompany\Accommodation\Room;
        $room2->room_number= 'A02';
        $room2->accommodation_id = $accommodation->id;
        $room2->save();

        //create the room array
        $rooms = [$room1,$room2];

    }

}
```

The seeder file sets up the very basic possible scenario. For initial testing, we don't need to even add every country, state, city, and location possible to the database; we simply need to add the essential information to create various scenarios. To create a new reservation; for example, we will create an instance of each of the user, country, state, city, location, and accommodation models, and then create two rooms, which are added to an array of rooms.

Let's create a repository for the reservation that will implement a very simple repository interface:

```
<?php
```

```
namespace MyCompany\Accommodation;

interface RepositoryInterface {
    public function create($attributes);
}
```

Now let's create `ReservationRepository`, which implements `RepositoryInterface`:

```
<?php

namespace MyCompany\Accommodation;

class ReservationRepository implements RepositoryInterface {
    private $reservation;

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

    public function create($attributes)
    {
        $this->reservation->create($attributes);
        return $this->reservation;
    }
}
```

Now, we will create the method that is needed to create the reservation and also to populate the pivot table, `reservation_room`:

```
public function create($attributes)
{

    $modelAttributes= array_except($attributes, ['rooms']);

    $reservation = $this->reservationModel-
>create($modelAttributes);
    if (isset($attributes['rooms']) ) {
        $reservation->rooms()->sync($attributes['rooms']);
    }
    return $reservation;
}
```

Tip

The `array_except()` Laravel helper is used to return the `attributes` array,

except for the `$rooms` array, which will be used for the `sync()` function.

Here, we set each attribute of the model to the attributes that are set in the method. We will need to add the method that will establish the many-to-many relationship between the reservations and rooms:

```
public function rooms(){
    return $this->belongsToMany('MyCompany\Accommodation\Room')-
>withTimestamps();
}
```

In this case, we need to add `withTimestamps()` to the relationship so that the timestamps will be updated, indicating when the relationship was saved in the `reservation_room` pivot table.

Database testing with PHPUnit

PHPUnit is well integrated with Laravel 5 as it was with Laravel 4, so it is rather easy to set up the testing environment. A good method for testing would be to use the SQLite database and to set it up to reside in the memory, but you need to modify the `config/database.php` file, as follows:

```
    'default' => 'sqlite',
       'connections' => array(
        'sqlite' => array(
            'driver'   => 'sqlite',
            'database' => ':memory:',
            'prefix'   => '',
        ),
    ),
```

Then, we need to modify the `phpunit.xml` file to set a `DB_DRIVER` environment variable:

```
<php>
        <env name="APP_ENV" value="testing"/>
        <env name="CACHE_DRIVER" value="array"/>
        <env name="SESSION_DRIVER" value="array"/>
        <env name="DB_DRIVER" value="sqlite"/>
</php>
```

Then, we need to modify the following line in the `config/database.php` file:

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

We modify the preceding line to match with the following line:

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

Now, we will set up PHPUnit to run our migrations on the in-memory `sqlite` database.

In the `tests` directory, there are two classes: a `TestCase` class that extends the `LaravelTestCase` class and an `ExampleTest` class that extends the `TestCase` class.

We need to add two methods to `TestCase` to perform the migrations, run the seeder, and then revert the database back to its original state:

```
<?php

class TestCase extends Illuminate\Foundation\Testing\TestCase {

    public function setUp()
    {
        parent::setUp();
        Artisan::call('migrate');
        Artisan::call('db:seed');
    }

    /**
    * Creates the application.
    *
    * @return \Illuminate\Foundation\Application
    */
    public function createApplication()
    {
        $app = require __DIR__.'/../bootstrap/app.php';
        $app->make('Illuminate\Contracts\Console\Kernel')-
>bootstrap();
        return $app;
    }

    public function tearDown()
    {
        Artisan::call('migrate:rollback');
    }
}
```

Now, we will create a PHPUnit test to verify that the data is being saved correctly in the database. We need to modify `tests/ExampleTest.php` to the following code:

```
<?php

class ExampleTest extends TestCase {

    /**
    * A basic functional test example.
    *
    * @return void
    */

public function testReserveRoomExample()
    {

        $reservationRepository = new
\MyCompany\Accommodation\ReservationRepository(
```

```
            new \MyCompany\Accommodation\Reservation());
        $reservationValidator = new
\MyCompany\Accommodation\ReservationValidator();
        $start_date = '2015-10-01';
        $end_date = '2015-10-10';
        $rooms = \MyCompany\Accommodation\Room::take(2)-
>lists('id')->toArray();
        if ($reservationValidator-
>validate($start_date,$end_date,$rooms)) {
            $reservation = $reservationRepository-
>create(['date_start'=>$start_date,'date_end'=>$end_date,'rooms'=
>$rooms,'reservation_number'=>'0001']);
        }

        $this-
>assertInstanceOf('\MyCompany\Accommodation\Reservation',$reserva
tion);
        $this->assertEquals('2015-10-01',$reservation-
>date_start);
        $this->assertEquals(2,count($reservation->rooms));
}
```

Running PHPUnit

To start PHPUnit, simply type the following command:

```
$ phpunit
```

The tests will be run. Since the `create` method of the `Reservation` class returns a reservation, we may use the `assertInstanceOf` method of PHPUnit to determine whether or not a reservation was created in the database. We can add any other assertions to make sure that the values saved are exactly what we intended. For example, we can assert that the start date is equal to `'2015-10-01'` and the size of the `room` array is equal to `two`. Together with the `testBasicExample()` method, we can ensure that a `GET` request to `"/"` returns a `200`. The PHPUnit results will look like this:



Notice that there were two dots to represent the tests. **OK** means that nothing failed, and we are told again that there were two tests and four assertions; one assertion in the example and the other three, which we have added to our `testReserveRoomExample` test. Had we tested that there were three rooms instead of two, PHPUnit would have produced the following output:

```
$ phpunit
PHPUnit 4.5.0 by Sebastian Bergmann and contributors.

Configuration read from /var/www/laravel.example/phpunit.xml

.
F
```

```
Time: 1.59 seconds, Memory: 10.75Mb

There was 1 failure:

1) ExampleTest::testReserveRoomExample
Failed asserting that 2 matches expected 3.

/var/www/laravel.example/tests/ExampleTest.php:24

FAILURES!
Tests: 2, Assertions: 4, Failures: 1.
```

Notice that instead of the second dot, we have an `F` for failure, and instead of `OK`, we're told that there was `1` failure. PHPUnit then lists which tests failed, and nicely tells us the line that I deliberately modified to be incorrect, as follows:

```
$this->assertEquals(3,count($reservationResult->rooms));
```

The preceding line is indeed incorrect:

```
Failed asserting that 2 matches expected 3.
```

Remember that `2` is the value of the count `($reservationResult->rooms)`.

Functional testing with Behat

While phpspec follows the BDD by specification and is useful for specification and design in isolation, its complimentary tool Behat is used for integration and functional tests. Since phpspec suggests to mock everything, database queries wouldn't actually be executed, as the database is outside the context of that method. Behat is a great tool to perform behavioral testing on a certain feature. While phpspec is already included among Laravel 5's dependencies, Behat will be installed as an external module.

The following command should be run to install and make Behat work with Laravel 5:

```
$ composer require behat/behat behat/mink behat/mink-extension
laracasts/behat-laravel-extension --dev
```

After running the composer update, Behat's functionality is added to Laravel. Next, a `behat.yaml` file should be added to the root of the Laravel project to specify which extensions are to be used.

Next, run the following command:

```
$ behat --init
```

This will create a `features` directory with a `bootstrap` directory inside it. A `FeaturesContext` class will also be created. Everything inside bootstrap will be run every time `behat` is run. This is useful to automatically run migrations and seeding.

The `features/bootstrap/FeaturesContext.php` file looks like this:

```
<?php

use Behat\Behat\Context\Context;
use Behat\Behat\Context\SnippetAcceptingContext;
use Behat\Gherkin\Node\PyStringNode;
use Behat\Gherkin\Node\TableNode;

/**
 * Defines application features from the specific context.
 */
class FeatureContext implements Context, SnippetAcceptingContext
{
```

```
    /**
     * Initializes context.
     *
     * Every scenario gets its own context instance.
     * You can also pass arbitrary arguments to the
     * context constructor through behat.yml.
     */
    public function __construct()
    {
    }
}
```

Next, the `FeatureContext` class needs to extend the `MinkContext` class, so the class definition line will need to be modified as follows:

```
class FeatureContext implements Context, SnippetAcceptingContext
```

Next, the `prepare` and `cleanup` methods will be added to the class in order to perform the migrations. We will add the `@BeforeSuite` and `@AfterSuite` annotations to tell Behat to perform the migration and seeding before each suite and migrate to rollback in order to restore the database to its original state after each suite. Using annotations in the doc-block will be discussed in Chapter 6, *Taming Complexity with Annotations*. Our class now is structured as follows:

```
<?php

use Behat\Behat\Context\Context;
use Behat\Behat\Context\SnippetAcceptingContext;
use Behat\Gherkin\Node\PyStringNode;
use Behat\Gherkin\Node\TableNode;

/**
 * Defines application features from the specific context.
 */
class FeatureContext implements Context, SnippetAcceptingContext
{
    /**
     * Initializes context.
     *
     * Every scenario gets its own context instance.
     * You can also pass arbitrary arguments to the
     * context constructor through behat.yml.
     */
    public function __construct()
    {
    }
     /**
```

```
     * @BeforeSuite
     */
     public static function prepare(SuiteEvent $event)
     {
        Artisan::call('migrate');
        Artisan::call('db:seed');

     }

     /**
     * @AfterSuite
     */
     public function cleanup(ScenarioEvent $event)
     {
        Artisan::call('migrate:rollback');
     }
}
```

Now, a feature file needs to be created. Create `reservation.feature` in the room directory:

```
Feature: Reserve Room
  In order to verify the reservation system
  As an accommodation reservation user
  I need to be able to create a reservation in the system
  Scenario: Reserve a Room
   When I create a reservation
         Then I should have one reservation
```

When `behat` is run as follows:

```
$ behat
```

The following output is produced:

```
Feature: Reserve Room
  In order to verify the reservation system
  As an accommodation reservation user
  I need to be able to create a reservation in the system

  Scenario: List 2 files in a directory #
features/reservation.feature:5
    When I create a reservation
    Then I should have one reservation

1 scenario (1 undefined)
2 steps (2 undefined)
0m0.10s (7.48Mb)
```

```
--- FeatureContext has missing steps. Define them with these
snippets:

    /**
     * @When I create a reservation
     */
    public function iCreateAReservation()
    {
        throw new PendingException();
    }

    /**
     * @Then I should have one reservation
     */
    public function iShouldHaveOneReservation()
    {
        throw new PendingException();
    }
```

Behat, as did phpspec, skillfully produces the output, showing you the methods that need to be created. Notice that camel case is used instead of snake case. This code should be copied in to the `FeatureContext` class. Notice that, by default, an exception is thrown.

Here, the RESTful API will be called, so the guzzle HTTP package will need to be added to the project:

```
$ composer require guzzlehttp/guzzle
```

Next, add an attribute to the class to hold the `guzzle` object. We will add a `POST` request to a RESTful resource controller to create a reservation and expect a 201 code. Notice that the return code is a string and needs to be casted to an integer. Next, a `get` is performed to return all of the reservations.

There should only be one reservation created, since the migration and seeding run every time:

```
<?php

use Behat\Behat\Context\Context;
use Behat\Behat\Context\SnippetAcceptingContext;
use Behat\Gherkin\Node\PyStringNode;
use Behat\Gherkin\Node\TableNode;
use Behat\MinkExtension\Context\MinkContext;
use Behat\Testwork\Hook\Scope\BeforeSuiteScope;
```

```
use Behat\Testwork\Hook\Scope\AfterSuiteScope;
use GuzzleHttp\Client;

/**
 * Defines application features from the specific context.
 */
class FeatureContext extends MinkContext implements Context,
SnippetAcceptingContext
{
    /**
     * Initializes context.
     *
     * Every scenario gets its own context instance.
     * You can also pass arbitrary arguments to the
     * context constructor through behat.yml.
     */
    protected $httpClient;

    public function __construct()
    {
        $this->httpClient = new Client();
    }
    /**
     * @BeforeSuite
     */
    public static function prepare(BeforeSuiteScope $scope)
    {
        Artisan::call('migrate');
        Artisan::call('db:seed');

    }

    /**
     * @When I create a reservation
     */
    public function iCreateAReservation()
    {
        $request = $this->httpClient-
>post('http://laravel.example/reservations',['body'=>
['start_date'=>'2015-04-01','end_date'=>'2015-04-
04','rooms[]'=>'100']]);
        if ((int)$request->getStatusCode()!==201)
        {
            throw new Exception('A successfully created status
code must be returned');
        }
    }

    /**
     * @Then I should have one reservation
```

```
     */
    public function iShouldHaveOneReservation()
    {
        $request = $this->httpClient-
>get('http://laravel.example/reservations');
        $arr = json_decode($request->getBody());
        if (count($arr)!==1)
        {
            throw new Exception('there must be exactly one
reservation');
        }
    }

    /**
     * @AfterSuite
     */
    public static function cleanup(AfterSuiteScope $scope)
    {
        Artisan::call('migrate:rollback');
    }
}

    /**
     * @When I create a reservation
     */
    public function iCreateAReservation()
    {
        $request = $this->httpClient-
>post('http://laravel.example/reservations',['body'=>
['start_date'=>'2015-04-01','end_date'=>'2015-04-
04','rooms[]'=>'100']]);
        if ((int)$request->getStatusCode()!==201)
        {
            throw new Exception('A successfully created status
code must be returned');
        }
    }
```

Now, to create `ReservationController`, use artisan from the command line:

```
$ php artisan make:controller ReservationsController
```

Here are the contents of the reservation controller:

```
<?php namespace MyCompany\Http\Controllers;

use MyCompany\Http\Requests;
use MyCompany\Http\Controllers\Controller;
use Illuminate\Http\Request;
```

```
use Symfony\Component\HttpFoundation\Response;
use MyCompany\Accommodation\ReservationRepository;
use MyCompany\Accommodation\ReservationValidator;
use MyCompany\Accommodation\Reservation;

class ReservationsController extends Controller {

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

    /**
    * Store a newly created resource in storage.
    *
    * @return Response
    */
    public function store()
    {
        $reservationRepository = new ReservationRepository(new
Reservation());
        $reservationValidator = new ReservationValidator();
        if ($reservationValidator-
>validate(\Input::get('start_date'),
        \Input::get('end_date'),\Input::get('rooms')))
        {
        $reservationRepository-
>create(['date_start'=>\Input::get('start_date'),'date_end'=>\Inp
ut::get('end_date'),'rooms'=>\Input::get('rooms')]);
        return response('', '201');
        }
    }
}
```

Lastly, add `ReservationController` to the `routes.php` file, which is located in `app/Http/routes.php`:

```
Route::resource('reservations','ReservationController');
```

Now, when `behat` is run, the result is as follows:

```
Feature: Reserve Room
  In order to verify the reservation system
  As an accommodation reservation user
```

```
  I need to be able to create a reservation in the system

  Scenario: Reserve a Room
    When I create a reservation         #
FeatureContext::iCreateAReservation()
    Then I should have one reservation  #
FeatureContext::iShouldHaveOneReservation()

1 scenario (1 passed)
2 steps (2 passed)
```

Summary

Configuring Laravel to create migration files from the existing schemas is also a useful framework for non-greenfield projects. By running both the migrations and seeding in the testing environment, each test can benefit from a completely clean version of the database, and the initial data that allows it to have "just enough" in the database to minimally verify that the software performs as it needs to. When legacy code needs to be ported to Laravel, PHPUnit can be used to test any existing functions. Behat provides a behavioral-based alternative, which can skillfully perform end-to-end testing.

We designed our classes using phpspec in an isolated environment, concentrating only on the business rules and client's requests while mocking things, such as the actual entities, such as rooms. We then verified that the actual queries were executed and saved in the database correctly by the use of a functional testing testing tool, PHPUnit. Finally, we used Behat to perform end-to-end testing.

In the next chapter, we'll see the creation of a RESTful API, the basic CRUD operations (create, read, update, and delete), and discuss some best practices.

Chapter 3. Building Services, Commands, and Events

In the first two chapters, we set up the basic structure of our accommodation reservation system. We designed our classes, created our database schema, and learned how to test them. Now we need to translate the business requirements into code.

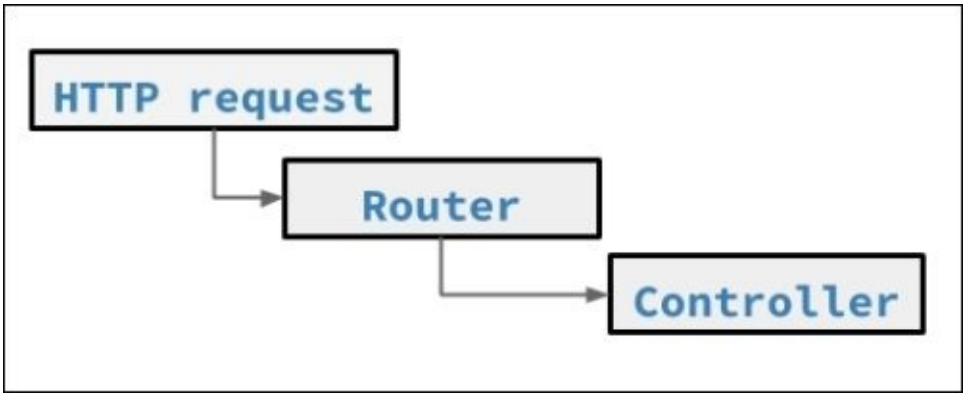
In this chapter, we will cover the following topics:

- Commands
- Events
- Command handlers
- Event handlers
- Queued event handlers
- Queued commands
- Console commands
- The command scheduler

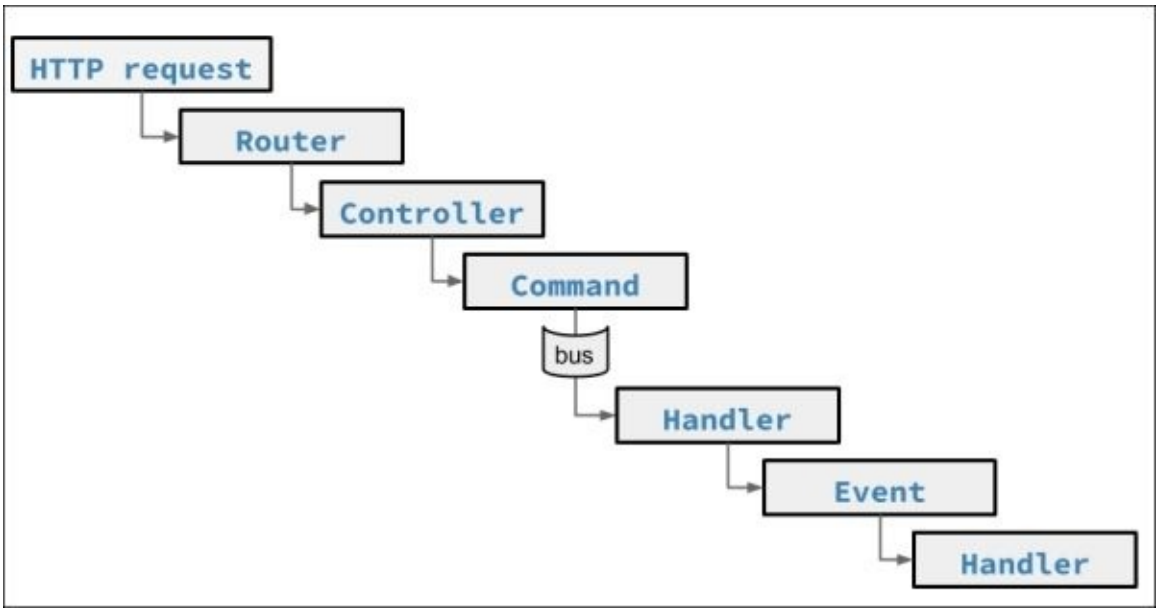
Request routing

As mentioned earlier, Laravel 5 has adopted the *command bus pattern*. Laravel 4 viewed commands as something to be executed from the command line, whereas in Laravel 5, a command can be used in any context, allowing excellent reuse of code.

The following is an example of the Laravel 4 HTTP request flow:



Here is an example of the Laravel 5 HTTP request flow:



The first image illustrates the Laravel 4 request flow. The request via HTTP was handled by the router, and then sent to the controller, where generally, we could then talk to either the repository or directory of the model. In Laravel 5, this is still possible; however, as shown in the second image, we can see that the ability to add additional blocks, layers, or modules allows us to separate the life cycle of the request into individual isolated pieces. Laravel 4 allowed us to put all of our code to handle the request inside the controller, while in Laravel 5, we are free to do the same, although now we are also able to easily separate the request into various pieces. Some of these concepts are derived from **Domain-driven Design** (**DDD**).

Inside the controller, the command is instantiated using the **Data Transfer Object** (**DTO**) paradigm. Then, the command is sent to the command bus, where it is handled by a handler class, which has two methods: `__construct()` and `handle()`. Inside the handler, we fire or instantiate an event. The event is likewise handled in the same way by an event handler method with two methods: `__construct()` and `handle()`.

The directory structure is very clean and looks like this:

```
/app/Commands
/app/Events/
/app/Handlers/
/app/Handlers/Commands
/app/Handlers/Events
/app/HTTP/Controllers
```

It's rather self-explanatory; the commands and events are in their respective directories, while the handlers for each have their own directories.

Note

Laravel 5.1 has changed the name of the `app/Commands` directory to `app/Jobs` to ensure that programmers do not get the concepts of the command bus and console commands mixed up.

User stories

The idea for the command component can easily derive from a user story or a task required by a user to achieve a goal. The most simple example would be to search for a room:

```
As a hotel website user,
I want to search for a room
so that I can select from a list of results.
```

User stories, deriving from the agile methodology, guarantee that the code written closely matches the business requirement. They often follow the pattern "As a… I want to… so that...". This defines the `actor`, the `intent`, and the `benefit`. It helps us plan how each task will be converted into code. In our example, the user stories can transform into tasks.

As a hotel website user, I would create a list of the following tasks:

1. As a hotel website user, I want to search for a room so that I can select a room from a list of results.
2. As a hotel website user, I want to reserve a room so that I can stay at the hotel.
3. As a hotel website user, I want to receive an e-mail with the reservation details so that I can have a copy of the reservation.
4. As a hotel website user, I want to be on a waiting list so that I can reserve a room when one becomes available.
5. As a hotel website user, I want to get notified of the room availability so that I can reserve a room.

User stories to code

The first task, searching for a room, would most likely be a RESTful call from a user or from an external service, so this task would be exposed to our controllers and thus, to our RESTful API.

The second task, reserving a room, is a similar action initiated by the user or the other service. This task may require the user to be logged in.

The third task could depend on the second task. This task requires an interaction with another process that sends the user a confirmation e-mail with the details of the booking. We can also write this as: *As a hotel website, I want to send an e-mail with the reservation details, so that he or she may have a copy of the reservation*.

The fourth task, getting placed on the waiting list, could be a command that is executed after the request to reserve a room is launched; in the case of another user reserving the room at the same time. It would most likely be called from the application itself, not the user, since the user has no knowledge of the real-time accommodation inventory. This could help us handle a race condition. Also, we should assume that when the website user is deciding which room to reserve, there is no locking mechanism on that room that would guarantee the availability. We could also write this as: *As a hotel website, I want to put a user on the waiting list so that they can be notified when a room is available*.

For the fifth task, as the user is put on a waiting list, the user could also be notified of the room as it becomes available. This action checks for the availability of the rooms, and then checks for any users on the waiting list. The user story can be rewritten as follows: *As a hotel website, I want to notify a waiting list user of the availability of the rooms so that he or she may reserve a room*. If a room becomes available, the first user on the waiting list will be notified of the availability via an e-mail. This command would be executed frequently, as if it is a cron job. Luckily, Laravel 5 has a new mechanism to allow commands to be executed at a given frequency.

It becomes apparent that if the user story has to be written with both using the website as the actor ("As a hotel website...") or the website user as the actor ("As a hotel website user..."), a command is useful and can be launched either from the RESTful API (user side) or from within the Laravel application itself.

Since our first task most likely involves an external service, we will create a route and also a controller to handle the request.

The controller

The first step involves the creation of a route, and the second step involves the creation of a controller.

Searching for the room

First, let's create a route in the `routes.php` file and map it to the `controller` method as follows:

```
Route::get('search', 'RoomController@search');
```

The request parameters, such as the start/end dates and location details will be as follows:

```
{
  "start_date": "2015-07-10"
  "end_date": "2015-07-17"
  "city": "London"
  "country": "England"
}
```

The search parameters will be sent as JSON-encoded objects. They will be sent as follows:

```
http://websiteurl.com/search?query={%22start_date%22:%222015-07-
10%22,%22end_date%22:%222015-07-
17%22,%22city%22:%22London%22,%22country%22:%22England%22}
```

Now, let's add a `search` method to our `room` controller to handle the JSON input in the case of a request that comes in as an object, as follows:

```
/**
* Search for a room in an accommodation
*/
public function search()
{
      json_decode(\Request::input('query'));
}
```

The request facade handles the input variable query, and then decodes its JSON structure into an object.

In Chapter 4, *Creating RESTful APIs*, we will complete the code for the `search`

method, but for now, we will simply create the architecture of this part of our RESTful API system.

Controller to command

For the second task, reserving the room, we'll create a command as we'll most likely need a follow up action, which we will enable via the publisher subscriber pattern. The publisher subscriber pattern is used to represent *publishers* that send messages and *subscribers* that listen to these messages.

Add the route to `routes.php` as follows:

```
Route::post('reserve-room', 'RoomController@store');
```

We map the post to the room controller's `store` method; this will create the reservation. Remember that we created the command like this:

```
$ php artisan make:commandReserveRoomCommand --handler
```

Our `ReserveRoomCommand` class looks like this:

```
<?php namespace MyCompany\Commands;

use MyCompany\Commands\Command;
use MyCompany\User;

class ReserveRoomCommand extends Command {

    public $user;
    public $rooms;
    public $start_date;
    public $end_date;

    /**
    * Create a new command instance.
    *
    * @return void
    */
    public function __construct(User $user, $start_date,
$end_date, $rooms)
    {
        $this->rooms = $rooms;
        $this->user = $user;
        $this->start_date = $start_date;
        $this->end_date = $end_date;
     }
```

```
}
```

We need to add the following attributes to the constructor:

```
    public $user;
    public $rooms;
    public $start_date;
    public $end_date;
```

Also, add the following assignments to the constructor:

```
        $this->rooms = $rooms;
        $this->user = $user;
        $this->start_date = $start_date;
        $this->end_date = $end_date;
```

This allows us to carry the values through.

Command to event

Now let's create an event. Use `artisan` to create an event, `RoomWasReserved`, which is to be fired when the room gets created:

```
$ phpartisan make:eventRoomWasReserved
```

The `RoomWasReserved` event class looks like the following code snippet:

```
<?php namespace MyCompany\Events;

use MyCompany\Accommodation\Reservation;
use MyCompany\Events\Event;
use MyCompany\User;

use Illuminate\Queue\SerializesModels;

class RoomWasReserved extends Event {

    use SerializesModels;

    private $user;
    private $reservation;

    /**
    * Create a new event instance.
    *
    * @return void
    */
    public function __construct(User $user, Reservation
```

```
$reservation)
    {
        $this->user = $user;
        $this->reservation = $reservation;
    }
}
```

We'll tell it to use the `MyCompany\Accommodation\Reservation` and `MyCompany\User` entities so that we can pass them to the constructor. Inside the constructor, we assign them to entities within the `event` object.

Now, let's fire the event from inside the command handler. Laravel provides you with a simple `event()` method as a convenience/helper method that will fire an event. We'll inject the `RoomWasReserved` event with the instantiated reservation and `user` as follows:

```
event(new RoomWasReserved($user, $reservation));
```

The ReserveRoomCommandHandler class

Our `ReserveRoomCommandHandler` class now instantiates a new reservation, uses the `createNew` factory method to inject the dependencies, and finally, fires the `RoomWasReserved` event as follows:

```
<?phpnamespace MyCompany\Handlers\Commands;

use MyCompany\Commands\ReserveRoomCommand;

use Illuminate\Queue\InteractsWithQueue;

class ReserveRoomCommandHandler {

    /**
    * Create the command handler.
    *
    * @return void
    */
    public function __construct()
    {
        //
    }

    /**
    * Handle the command.
    *
    * @paramReserveRoomCommand  $command
    * @return void
```

```
    */
    public function handle(ReserveRoomCommand $command)
    {

        $reservationValidator = new
\MyCompany\Accommodation\ReservationValidator();

        if ($reservationValidator->validate($command-
>start_date,$command->end_date,$command->rooms)) {
              $reservation =
                $reservationRepository->create(
                ['date_start'=>$command->$command→start_date,
                'date_end'=>$command->end_date,
                'rooms'=>$command->'rooms']);
        }
    $reservation = new
      event(new RoomWasReserved($command->user,$reservation));
    }
}
```

Event to handler

Now, we need to create the event handler. As you would have expected, the Artisan provides a convenient way of doing this, although the syntax is a bit different. This time, strangely, the word *make* doesn't appear in the phrase:

```
$ php artisan handler:eventRoomReservedEmail --
event=RoomWasReserved
    <?php namespace MyCompany\Handlers\Events;

    use MyCompany\Events\RoomWasReserved;

    use Illuminate\Queue\InteractsWithQueue;
    use Illuminate\Contracts\Queue\ShouldBeQueued;

    class RoomReservedEmail {

        /**
        * Create the event handler.
        * @return void
        */
        public function __construct()
        {
        }

        public function handle(RoomWasReserved $event)
        {
            //TODO: send email to $event->user
            //TODO: with details about $event->reservation;
```

```
        }
    }
```

Now we need to connect the event to its listener. We will edit the `app/Providers/EventServiceProvider.php` file as follows:

```
protected $listen = [
    'MyCompany\Events\RoomWasReserved' => [
      'MyCompany\Handlers\Events\RoomReservedEmail',
      ],
    ];
```

As shown in the preceding code snippet, we will add the key-value pair to the `$listen` array. The full path, as shown, is needed for the key, the event name, and the array of handlers. In this case, we only have one handler.

Queued event handlers

If we would like to not have the event handled immediately, but rather, put into the queue, we can add `–queued` to the create command as follows:

```
$ php artisan handler:eventRoomReservedEmail --
event=RoomWasReserved --queued
```

```
    <?php namespace MyCompany\Handlers\Events;

    use MyCompany\Events\RoomWasReserved;

    use Illuminate\Queue\InteractsWithQueue;
    use Illuminate\Contracts\Queue\ShouldBeQueued;

    class RoomReservedEvent implements ShouldBeQueued {

    use InteractsWithQueue;

    public function __construct()
    {
        //
    }

    use Illuminate\Contracts\Queue\ShouldBeQueued;
```

This interface tells Laravel that the event handler should be queued and not executed synchronously:

```
use Illuminate\Queue\InteractsWithQueue;
```

This trait allows us to interact with the queue to be able to do tasks, such as delete the job.

The waiting list command

For the fourth task, being placed on a waiting list, we'll need to create another command that would be called from inside the reservation controller. Once again, using Artisan, we can easily create the command and its corresponding event as follows:

```
$ php artisan make:commandPlaceOnWaitingListCommand
$ php artisan make:eventPlacedOnWaitinglist
```

Now, in our reservation controller, we would add the check for `roomAvailability` and then dispatch the `PlaceOnWaitinglist` command as follows:

```
public function store()
    {
    …
    …
        if ($roomAvailable) {
            $this->dispatch(
              new ReserveRoomCommand( $start_date, $end_date,
$rooms)
            );
        } else {
            $this->dispatch(
              new PlaceOnWaitingListCommand($start_date,
$end_date, $rooms)
            );
        }
    …
```

The queued commands

We can easily queue the commands by adding `queued` to the `create` command:

```
$ php artisan make:commandReserveRoomCommand --–handler --queued
```

This will use whichever queuing system is available, such as beanstalkd, and not immediately run the command. Instead, it will be placed in the queue and run later. We'll need to add an interface to the `Command` class:

```
Illuminate\Contracts\Queue\ShouldBeQueued
```

In this case, the `ReserveRoomCommand` class will look like this:

```
<?php namespace MyCompany\Commands;

use MyCompany\Commands\Command;

use Illuminate\Queue\SerializesModels;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldBeQueued;

class MyCommand extends Command implements ShouldBeQueued {

        use InteractsWithQueue, SerializesModels;

        /**
         * Create a new command instance.
         *
         * @return void
         */
        public function __construct()
        {
                //
        }

}
```

Here, we can see that the `InteractsWithQueue` and `ShouldBeQueued` classes have been included, and the `ReserveRoomCommand` class extends the command and implements the `ShouldBeQueued` class. Another interesting feature is `SerializesModels`. This will serialize any models, which are passed in, to be available later.

The console command

For the fifth task, let's create a `console` command, which will be executed very often:

```
$ php artisan make:consoleManageWaitinglist
```

This will create a command that can be executed from the Artisan command-line tool. If you have used Laravel 4, you may be familiar with this type of command. These commands are stored in the `Console/Commands/` directory.

To let Laravel know about it, we will need to add it to `app/Console/Kernel.php` in the `$commands` array:

```
protected $commands = [
    'MyCompany\Console\Commands\Inspire',
    'MyCompany\Console\Commands\ManageWaitinglist',
    ];
```

The contents will look like this:

```
<?php namespace MyCompany\Console\Commands;

use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputOption;
use Symfony\Component\Console\Input\InputArgument;

class ManageWaitinglist extends Command {

    /**
    * The console command name.
    *
    * @var string
    */
    protected $name = 'command:name';

    /**
    * The console command description.
    *
    * @var string
    */
    protected $description = 'Command description.';

    /**
    * Create a new command instance.
    *
```

```
    * @return void
    */
    public function __construct()
    {
        parent::__construct();
    }

    /**
    * Execute the console command.
    *
    * @return mixed
    */
    public function fire()
    {
        //
    }

    /**
    * Get the console command arguments.
    *
    * @return array
    */
    protected function getArguments()
    {
        return [
          ['example', InputArgument::REQUIRED, 'An example
argument.'],
        ];
    }

    /**
    * Get the console command options.
    *
    * @return array
    */
    protected function getOptions()
    {
        return [
          ['example', null, InputOption::VALUE_OPTIONAL, 'An
example option.', null],
        ];
    }
}
```

The `$name` attribute is what will be called from Artisan. For example, if we set the following:

```
protected $name = 'manage:waitinglist';
```

Then, by running the following command, we can manage the waiting list:

```
$ php artisan manage:waitinglist
```

The `getArguments()` and `getOptions()` methods are similar methods with the same signature, but have different uses.

The `getArguments()` method specifies an array of arguments that must be used to launch the command. The `getOptions()` methods are specified with – and can be `optional`, `repeated`, and with the `VALUE_NONE` option, they can be simply used as flags.

We will write the command's main code inside the `fire()` method. If we want to dispatch a command from within this command, we'll add the `DispatchesCommands` trait to the class as follows:

```
    use DispatchesCommands;

    <?php namespace MyCompany\Console\Commands;

    use Illuminate\Console\Command;
    use Illuminate\Foundation\Bus\DispatchesCommands;
    use Symfony\Component\Console\Input\InputOption;
    use Symfony\Component\Console\Input\InputArgument;

    class ManageWaitinglist extends Command {

        use DispatchesCommands;

        /**
        * The console command name.
        * @var string
         */
        protected $name = 'manage:waitinglist';

        /**
        * The console command description.
        * @var string
        */
        protected $description = 'Manage the accommodation
waiting list.';

        /**
        * Create a new command instance.
        *
        * @return void
        */
```

```
        public function __construct()
        {
            parent::__construct();
        }

        /**
        * Execute the console command.
        * @return mixed
        */
        public function fire()
        {
            // TODO: write business logic to manage waiting list
            if ($roomIsAvailableFor($user)) {
                $this->dispatch(new ReserveRoomCommand());
            }
        }

        /**
        * Get the console command arguments.
        * @return array
        */
        protected function getArguments()
        {
            return [];
        }

        /**
        * Get the console command options.
        * @return array
        */
        protected function getOptions()
        {
            return [];
        }
}
```

The command scheduler

Now, we will schedule this command to run every 10 minutes. Traditionally, this was done by creating a cron job to execute the Laravel console's command. Now, Laravel 5 provides a new mechanism to do this—the command scheduler.

The new `artisan` command is run as follows:

```
$ php artisan schedule:run
```

By simply adding this command to the cron, Laravel will automatically run all of the commands that are in the `Kernel.php` file.

The commands need to be added to the `Schedule` function as follows:

```
protected function schedule(Schedule $schedule)
    {
        $schedule->command('inspire')
             ->hourly();
        $schedule->command('manage:waitinglist')
            ->everyFiveMinutes();

    }
```

The `inspire` command is a sample command provided by Laravel to demonstrate the functionality. We will simply add our command. This will call the `manage:waitinglist` command every 5 minutes—it couldn't be much easier than that.

Now we need to modify the `crontab` file to have Artisan run the scheduler.

The `crontab` is a file that contains commands to be run at certain times. To modify this file, type the following command:

```
$ sudo crontab -e
```

We will use `vi` or whichever the assigned editor is to modify the `cron` table. Adding the following line will tell `cron` to run the scheduler every minute:

```
* * * * * php /path/to/artisan schedule:run 1>> /dev/null 2>&1
```

Summary

Laravel has evolved in just two short years, moving away from CodeIgniter's Model-View-Controller paradigm to adopt modern Domain-driven design's command bus and publisher subscriber event listener pattern. Whether or not to use these patterns will depend on the amount of separation that is desired between each layer. Certainly, even using a self-handling command is a start toward creating completely independent blocks of code, which promotes the code into a separate handler class, carrying the separation-of-concerns principle even further. By reducing the amount of code that resides inside the controller, the command becomes even more important.

We have not yet even written the code for each user story to interact with the database, and we have only seeded and tested the database, but the structure is beginning to become very well designed; each class having a very meaningful name and being organized into a useful directory structure.

In the next chapter, we will fill in the details of how the RESTful controller will accept input from another system or from the frontend of the website, and then how the model's attributes will be returned to the user to create the interface.

Chapter 4. Creating RESTful APIs

If there is one single core feature that demonstrates Laravel's superiority, it would be the ability to quickly and easily create a RESTful API. With the arrival of Laravel 5, several new features have been added; however, the ability to create application models and controllers via the Artisan command-line tool remains the most useful feature.

This feature is what initially encouraged me and so many others to abandon frameworks such as CodeIgniter, which at the time that Laravel 4 was in beta, did not natively have the same integrated functionality. Laravel provides the basic CRUD methods: create, read, update, delete, and also lists all.

Requests that arrive via HTTP to a Laravel URL are managed through their verbs and subsequently, the `routes.php` file, which is located at `app/Http/routes.php`. There are two ways in which the requests are handled. One way is that the request is handled directly via a closure, and the code is entirely inside the `routes` file. Another way is that it routes the request to a controller, where a method will be executed.

Also, the basic paradigm used is convention-over-configuration, where the method names are ready to handle the various requests, without too much extra effort.

RESTful APIs in Laravel

The list of RESTful API requests handled by the RESTful API are as follows:

	HTTP VERB	Function	URL
1	`GET`	This lists all accommodations	`/accommodations`
2	`GET`	This shows (reads) a single accommodation	`/accommodations/{id}`
3	`POST`	This creates a new accommodation	`/accommodations`
4	`PUT`	This entirely modifies (updates) an accommodation	`/accommodations/{id}`
5	`PATCH`	This partially modifies (updates) an accommodation	`/accommodations/{id}`
6	`DELETE`	This deletes an accommodation	`/accommodations/{id}`

Most RESTful API best practices suggest using the plural form of the model name. Laravel's documentation uses the singular format. Most practices agree that consistent plural naming, that is, `/accommodations/{id}` refers to a single accommodation and `/accommodations` refers to more than one accommodation, both using the plural form are preferred over a mixed, but grammatically correct `/accommodation/{id}` (singular form) and `/accommodations` (plural form).

Essential CRUD

For simplicity, I have numbered each of the rows. The first and second items represent the *read* part of CRUD.

The first item, which is a `GET` call to the plural form of the model name, is rather simple; it displays all of the items. Sometimes, this is called a *list* to differentiate it from the *read* of a single record. Adding a *list* would thus expand the acronym to CRUDL. They could be paginated or require authorization.

The second item, also a `GET` call, adds the ID of the model to the end of the URL, displaying a single model with that corresponding ID. This could also require authentication but not paging.

The third item represents the *create* part of CRUD. It uses the `POST` verb to create a new model. Note that the URL format is the same as the first item; this demonstrates the importance of the verb to distinguish between the actions.

The fourth, fifth, and sixth items use the new `HTTP` verbs that were not supported by all browsers. Whether or not the verbs are supported, JavaScript libraries and frameworks, such as jQuery, will send the verb in a way that Laravel can properly handle.

The fourth item is the `update` part of CRUD and updates the model using the `PUT` verb. Note that it has the same URL format as the second, as it needs to know which model to update. It is also idempotent, which means that the entire model must be updated.

The fifth item is similar to the fourth item; it updates the model, but uses the `PATCH` verb. This is used to indicate that the model will be partially modified, which means that one or more of the model's attributes have to be changed.

The sixth item deletes a single model and thus requires the model's ID, using the self-explanatory `DELETE` verb.

Bonus features

Laravel adds two additional methods that are not usually part of a standard RESTful API. A `GET` method on the model URL, adding `create` is used to display a form to create the model. A `GET` method on the model URL with its ID, adding `edit` is used to display a form to create the model. These two functions are useful for providing a URL that will load a form, even though this type of usage is not a standard RESTful:

HTTP VERB	**Function**	**URL**
`GET`	This displays an accommodation creation form	`/accommodations/create`
`GET`	This displays an accommodation modification/update form	`/accommodations/{id}/edit`

Controller creation

To create a controller for the accommodations, the following Artisan command is used:

```
$ php artisan make:controller AccommodationsController
```

```
    <?php namespace MyCompany\Http\Controllers;

    use MyCompany\Http\Requests;
    use MyCompany\Http\Controllers\Controller;
    use Illuminate\Http\Request;

    class AccommodationController extends Controller {

        /**
        * 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)
    {
    }
}
```

CRUD(L) by example

We have seen this controller before, but here are a few examples. The single most simple example of a RESTful call would be as shown in the following sections.

cRudl – read

Create a `GET` call to `http://www.hotelwebsite.com/accommmodations/1`, where `1` would be the ID of the room:

```
/**
 * Display the specified resource.
 *
 * @param  int  $id
 * @return Response
 */
public function show($id)
{
    return \MyCompany\Accommodation::findOrFail($id);
}
```

This would return a single model as a JSON-encoded object:

```
{
    "id": 1,
    "name": "Hotel On The Hill","description":"Lorem ipsum dolor
sit amet, consectetur adipiscing elit.",
    "location_id": 1,
    "created_at": "2015-02-08 20:13:10",
    "updated_at": "2015-02-08 20:13:10",
    "deleted_at": null
}
```

crudL – list

Create a GET call to `http://www.hotelwebsite.com/accommmodations`.

This is similar to the preceding code, yet slightly different:

```
/** Display a listing of the resource.
    * @return Response
 */
public function index()
{
    return Accommodation::all();
}
```

This would return all of the models, automatically encoded as JSON objects; there is nothing else that is required. Formatting has been added so that the JSON results are more easily readable, but basically, the entire model is returned:

```
[{
    "id": 1,
    "name": "Hotel On The Hill","description":"Lorem ipsum dolor
sit amet, consectetur adipiscing elit.",
    "location_id": 1,
    "created_at": "2015-02-08 20:13:10",
    "updated_at": "2015-02-08 20:13:10",
    "deleted_at": null
}
{   "id": 2,
    "name": "Patterson Place",
    "description": "Lorem ipsum dolor sit amet, consectetur
adipiscing elit.",
    "location_id": 2,
    "created_at": "2015-02-08 20:15:02",
    "updated_at": "2015-02-08 20:15:02",
    "deleted_at": null
},
{
    "id": 3,
    "name": "Neat and Tidy Hotel",
    "description": "Lorem ipsum dolor sit amet, consectetur
adipiscing elit.",
    "location_id": 3,
    "created_at": "2015-02-08 20:17:34",
    "updated_at": "2015-02-08 20:17:34",
    "deleted_at": null
}
]
```

Tip

The `deleted_at` field is a soft delete or the recycle bin mechanism. It is either `null` for not deleted or a `date/time` stamp for deleted.

Pagination

To add pagination, simply substitute `all()` with `paginate()`:

```
public function index()
{
    return Accommodation::paginate();
}
```

The results will now look like this. The eloquent collection array is now moved inside a `date` attribute:

```
{"total":15,
"per_page":15,
"current_page":1,
"last_page":1,
"next_page_url":null,
"prev_page_url":null,
"from":1,
"to":15,
"data":[{"id":9,
"name":"Lovely Hotel",
"description":"Lovely Hotel Greater Pittsburgh",
….
```

Crudl – create

Create a `POST` call to `http://www.hotelwebsite.com/accommmodations`.

To create a new model, a `POST` call will be sent to `/accommodations`. A JSON would be sent from the frontend as follows:

```
{
    "name": "Lovely Hotel",
    "description": "Lovely Hotel Greater Pittsburgh",
    "location_id":1
}
```

The `store` function might look something like this:

```
public function store()
{
    $input = \Input::json();
    $accommodation = new Accommodation;
    $accommodation->name = $input->get('name');
    $accommodation->description = $input->get('description');
    $accommodation->location_id = $input->get('location_id');
    $accommodation->save();
    return response($accommodation, 201)
;
}
```

Tip

`201` is the HTTP status code (`HTTP/1.1 201 created`) for `created`.

In this example, we returned the model as a JSON-encoded object. The object will include the ID that was inserted:

```
{
    "name":"Lovely Hotel",
    "description":"Lovely Hotel Greater Pittsburgh",
    "location_id":1,
    "updated_at":"2015-03-13 20:48:19",
    "created_at":"2015-03-13 20:48:19",
    "id":26
}
```

crUdl – update

Create a `PUT` call to `http://www.hotelwebsite.com/accommmodations/1`, where `1` is the ID to be updated:

```
/**
    * Update the specified resource in storage.
    *
    * @param  int  $id
    * @return Response
    */
    public function update($id)
    {
        $input = \Input::json();
        $accommodation =
\MyCompany\Accommodation::findOrFail($id);
        $accommodation->name = $input->get('name');
        $accommodation->description = $input->get('description');
        $accommodation->location_id = $input->get('location_id');
        $accommodation->save();
        return response($accommodation, 200)
            ->header('Content-Type', 'application/json');
    }
```

To update an existing model, the code is exactly the same as we used earlier, except that the following line is used to find the existing model:

```
$accommodation = Accommodation::find($id);
```

The `PUT` verb would be sent to `/accommodations/{id}`, where `id` would be the numeric ID of the accommodations table.

cruDl – delete

To delete a model, create a `DELETE` call to `http://www.hotelwebsite.com/accommmodation/1`, where `1` is the ID to be deleted:

```
/**
 * Remove the specified resource from storage.
 *
 * @param  int  $id
 * @return Response
 */
public function destroy($id)
{
    $accommodation = Accommodation::find($id);
    $accommodation->delete();
    return response('Deleted.', 200)
;
}
```

Tip

There seems to be some disagreement about what the proper status code should be for a deleted model.

Model binding

Now, we can use a technique called *model binding* to clean up the code even more:

```
public function boot(Router $router)
{
    parent::boot($router);
    $router->model('accommodations', '\MyCompany\Accommodation');
}
```

In `app/Providers/RouteServiceProvider.php`, add the `$router->model()` method that accepts the route as the first argument and the model that will be bound as the second argument.

Read revisited

Now, our `show` controller method looks like this:

```
public function show(Accommodation $accommodation)
{
    return $accommodation;
}
```

When `/accommodations/1` is called, for example, the model that corresponds to that ID will be injected into the method, allowing us to substitute the find method.

List revisited

Similarly, for the `list` method, we inject the type-hinted model as follows:

```
public function index(Accommodation $accommodation)
{
    return $accommodation;
}
```

Update revisited

Likewise, the `update` method now looks like this:

```
public function update(Accommodation $accommodation)
{
    $input = \Input::json();
    $accommodation->name = $input->get('name');
    $accommodation->description = $input->get('description');
    $accommodation->location_id = $input->get('location_id');
    $accommodation->save();
    return response($accommodation, 200)
    ->header('Content-Type', 'application/json');
}
```

Delete revisited

Also, the `destroy` method looks like this:

```
public function destroy(Accommodation $accommodation)
{
    $accommodation->delete();
    return response('Deleted.', 200)
        ->header('Content-Type', 'text/html');
}
```

Moving beyond CRUD

If one of the requirements of the software application to be built is able to search for an accommodation, then we can easily add a search function. The search function will find accommodations by using a `name` string. One way to do this is to add a route to the `routes.php` file. This will map a `GET` call to search for a new `search()` function contained within `AccommodationsController`:

```
Route::get('search', 'AccommodationsController@search');
Route::resource('accommodations', 'AccommodationsController');
```

Tip

In this case, the `GET` method would be preferred instead of the `POST` method, as it can be bookmarked and recalled later.

Now, we will write our search function:

```
public function search(Request $request, Accommodation
$accommodation)
{
    return $accommodation
        ->where('name',
          'like',
          '%'.$request->get('name').'%')
        ->get();
    }
```

There are several mechanisms here:

- The `Request` object that contains the variables from the `GET` request is type-hinted and then injected into the search function
- The `Accommodation` model is type-hinted and then injected into the `search` function
- The `where()` method from the fluent query builder is called on the eloquent model `$accommodation`
- The `name` parameter is used from the `request` object
- The `get()` method is used to actually perform the SQL query

  ### Tip

  Note that some of the query builder and eloquent methods return an instance of the query builder, while the others execute the query and return the result.

The `where()` method returns an instance of the query builder, while the `get()` method executes the query.

- The resulting eloquent collection is returned and automatically encoded into JSON

The `GET` request, therefore, is as follows:

```
http://www.hotelwebsite.com/search-accommodation?name=Lovely
```

The resultant JSON would look something like this:

```
[{"id":3,
"name":"Lovely Hotel",
"description":"Lovely Hotel Greater Pittsburgh",
"location_id":1,
"created_at":"2015-03-13 22:00:23",
"updated_at":"2015-03-13 22:00:23",
"deleted_at":null},
{"id":4,
"name":"Lovely Hotel",
"description":"Lovely Hotel Greater Philadelphia",
"location_id":2,
"created_at":"2015-03-11 21:43:31",
"updated_at":"2015-03-11 21:43:31",
"deleted_at":null}]
```

Nested controllers

Nested controllers is a new feature in Laravel 5 and is used to handle all of the RESTful actions that deal with relationships. For example, we can take advantage of this feature for the relationship between accommodations and rooms.

The relationship between accommodation and room is as follows:

- An accommodation may have one or more rooms (one-to-many)
- A room belongs to one and only one accommodation (one-to-one)

In our models, we will now write the code to enable the one-to-one and one-to-many relationships to be skillfully handled by Laravel.

Accommodation hasMany rooms

First, we will add the code that is needed by the `Accomodation.php` file that represents the `accommodation` model as follows:

```
class Accommodation extends Model {
    public function rooms(){
        return $this->hasMany('\MyCompany\Accommodation\Room');
    }
}
```

The `rooms()` method creates an easy way to access the relationship from inside the accommodation model. The relation states that "the accommodation *hasMany* rooms". The `hasMany` function, when residing inside the `Accommodation` class, without any additional parameters, expects a column named `accommodation_id` to exist in the `Room` model's table, which in this case is `rooms`.

Room belongsTo accommodation

Now, we will add the code that is needed by the `Room.php` file that represents the `Room` model:

```
class Room extends Model
{
    public function accommodation(){
        return $this->belongsTo('\MyCompany\Accommodation');
    }
}
```

This code states that "a room *belongsTo* an accommodation". The *belongsTo* method inside the `Room` class, without any additional parameters, expects a field in the `room` model's table; in this case, `rooms`, named `accommodation_id`.

Tip

If the tables in the application database have followed the active record conventions, then most of the eloquent relation functionalities will automatically function. All of the parameters can be easily configured.

The command to create a nested controller is as follows:

```
$php artisan make:controller AccommodationsRoomsController
```

Then, the following line would be added to the `app/Http/routes.php` file:

```
Route::resource('accommodations.rooms',
'AccommodationsRoomsController');
```

To display the routes created, the following command should be executed:

```
$php artisan route:list
```

The following table lists the HTTP verbs and their functions:

	HTTP verb	Function	URL
1	`GET`	This shows the accommodation	`/accommodations/{accommodations}/rooms`

		and room relations	
2	GET	This shows an accommodation and room relation	/accommodations/{accommodations}/rooms/{roo
3	POST	This creates a new accommodation and room relation	/accommodations/{accommodations}/rooms
4	PUT	This entirely modifies (updates) an accommodation and room relation	/accommodations/{accommodations}/rooms/{roo
5	PATCH	This partially modifies (updates) an accommodation and room relation	/accommodations/{accommodations}/rooms/{roo
6	DELETE	This deletes an accommodation and room relation	/accommodations/{accommodations}/rooms/{roo

Eloquent relations

A nice mechanism used to illustrate an Eloquent relation directly inside the controller is performed through the use of a **nested relation**, where two models are connected firstly through the route and secondly through their controller method's parameters via model dependency injection.

Nested update

Let's investigate the `update`/`modify` `PUT` nested controller command. The URL looks like this:
`http://www.hotelwebsite.com/accommodations/21/rooms/13`.

Here, `21` would be the ID of the accommodation and `13` would be ID of the room. The parameters are the type-hinted models. This allows us to easily update the relationship as follows:

```
public function update(Accommodation $accommodation, Room $room)
{
    $room->accommodation()->associate($accommodation);
    $room->save();
}
```

Nested create

Similarly, it is easy to perform the nested `create` operation with a `POST` body to `http://www.hotelwebsite.com/accommodations/21/rooms`. The `POST` body is a JSON formatted object:

```
{"roomNumber":"123"}
```

Note that there is no ID needed for the room since we are creating it:

```
public function store(Accommodation $accommodation)
{
    $input = \Input::json();
    $room = new Room();
    $room->room_number = $input->get('roomNumber');
    $room->save();
    $accommodation->rooms()->save($room);
}
```

Eloquent model casting

Models are returned in the JSON format as they are represented in the database. Often, model attributes, which are Boolean in nature, are represented by `0` and `1` for `true` and `false`, respectively. It may be, in this case, more convenient to return a real `true` and `false` to the RESTful call's return object.

In Laravel 4, this was done using **accessors**. If the value was `$status`, the method would be defined as follows:

```
public function getStatusAttribute($value){
    //do conversion;
}
```

In Laravel 5, this process is much easier, thanks to a new feature called model casting. To apply this technique, simply add a protected key and a value array called `$casts` to the model as follows:

```
class Room extends Model
{
    protected $casts =
['room_number'=>'integer','status'=>'boolean'];
    public function accommodation(){
        return $this->belongsTo('\MyCompany\Accommodation');
    }
}
```

In this example, `room_number` is a string, but we want to return an integer. Status is a tiny integer, but we want to return a Boolean value. Casting these two values in the model will modify the resultant JSON in the following manner:

```
{"id":1,
"room_number": "101",
"status": 1,
"created_at":"2015-03-14 09:25:59",
"updated_at":"2015-03-14 19:03:03",
"deleted_at":null,
"accommodation_id":2}
```

The preceding code will now change as follows:

```
{"id":1,
"room_number": 101,
"status": true,
```

"created_at":"2015-03-14 09:25:59",
"updated_at":"2015-03-14 19:03:03",
"deleted_at":null,
"accommodation_id":2}

Route caching

Laravel 5 has a new mechanism for caching the routes as the `routes.php` file can easily grow very large and will quickly slow down the request process. To enable the caching mechanism, type the following `artisan` command:

```
$ php artisan route:cache
```

This creates another `routes.php` file in `/storage/framework/routes.php`. If this file exists, then it is used instead of the `routes.php` file, which is located in `app/Http/routes.php`. The structure of the file is as follows:

```
<?php

/*
|--------------------------------------------------------------------------
| Load The Cached Routes
|
…
*/

app('router')->setRoutes(
unserialize(base64_decode('TzozNDoiSWxsdW1pbmF0ZVxSb3V0aW5nXFJvdX
RlQ29sbGVjdGlvbiI6NDp7czo5OiIAKgByb3V0ZXMiO2E6Njp7czozOiJHRVQiO2E
6M
…
...
VyQGluZGV4IjtzOjk6Im5hbWVzcGFjZSI7czoyNjoiTXlDb21wYWbXBhbnlcSHR0c
FxDb250cm9sbGVyc1xIb3RlbENvbnRyb2xsZXJAZGVzdHJveSI7cjo4Mzg7fX0=')
)
);
```

Notice that an interesting technique is used here. The routes are serialized, then base64 is encoded. Obviously, to read the routes, the reverse is used, `base64_decode()`, and then `unserialize()`.

If the `routes.php` cached file exists, then every time a change is made to the `routes.php` file, the route cache `artisan` command must be executed. This will clear the file and then recreate it. If you later decide to no longer use this mechanism, then the following `artisan` command can be used to eliminate the file:

```
$ php artisan route:clear
```

Laravel is useful for building several distinctly different types of applications. When building traditional web applications, there is often a tight integration between the controllers and the views. It is also useful when building an app that can be used on a smartphone. In this case, the frontend will be created for the smartphone's operating system using another programming language and/or framework. In this case, only the controllers and model will most likely be used. In either case, however, having a well-documented RESTful API is an essential part of a well-designed modern software.

Nested controllers helps developers right away to read the code—it is an easy way to understand that the particular controller deals with the "nesting" or the concept that one class is related another.

Type-hinting the models and objects into the controller also improves the readability and, at the same time, reduces the amount of code necessary to perform the basic operations on the objects.

Also, eloquent model casting creates an easy way to transform the attributes of a model, without having to rely on external packages or tedious accessor functions, as was the case in Laravel 4.

Now it is rather clear to us why Laravel is becoming the choice of many developers. Learning and repeating some of the steps illustrated in this chapter will allow a RESTful API to get created in under an hour for a small-to-medium size program.

Summary

A RESTful API provides an easy way to expand the program in the future and also integrates with third-party programs and software that exist within a company that might need to communicate with the application. The RESTful API is the front-most shell of the inner part of the program and provides the bridge between the outside world and the application itself. The inner part of the program will be where all of the business logic and database connections will reside, so fundamentally, the controllers simply have the job of connecting the routes to the application.

Laravel follows the RESTful best practices, thus documenting the API should be easy enough for other developers and third-party integrators to understand. Laravel 5 has brought a few features in to the framework to enable the code to be more readable.

In future chapters, middleware will be discussed. Middleware adds various "middle" layers between the route and the controller. Middleware can provide features such as authentication. Middleware will enrich, protect, and help organize the routes into logical and functional groups.

We will also discuss DocBlock annotations. Annotations, while not natively supported in PHP, can be enabled via a Laravel community package. Then, inside the DocBlock of the controller and controller functions, the routing for each controller is automatically created, without having to actually modify the `app/Http/routes.php` file. This is another great community concept that Laravel easily adapts to, in the same manner as phpspec and Behat.

Chapter 5. Using the Form Builder

In this chapter, you will learn how to use Laravel's form builder. The form builder will be demonstrated to facilitate the building of the following elements:

- Form (open and close)
- Label
- Input (text, HTML5 password, HTML5 e-mail, and so on.)
- Checkbox
- Submit
- Anchor tags (href links)

Finally, we'll see an example of how to use the form builder to create the month, date, and year selection elements for the accommodations reservation software form, and how to create a macro to reduce the code duplication.

History

The form builder package in Laravel 4 is called HTML. This was used to help you create HTML, particularly developers who also have to perform web designer duties but prefer to use Laravel facades and helper methods. For example, the following Laravel facade `select()` method, where the options for the language, British and American English in this example, are passed as an array parameter:

```
Form::select('language', ['en-us' => 'English (US)','en-gb' =>
'English (UK)']);
```

This can be used as an alternative to the standard HTML, which requires much more repetitious code, as shown in the following code:

```
<select name="language">
    <option value="en-us">English (US)</option>
    <option value="en-gb">English (UK)</option>
</select>
```

Since frameworks are constantly evolving, they need to adapt to fulfill the needs of most of their users. Also, whenever possible, they should continue to be more efficient. In some cases, this means rewriting or refactoring pieces of the framework, adding features, or even *removing* them.

As strange as it may seem, there are several valid reasons for removing the features. The following is a list of reasons for removing packages:

- To ease the burden and quantity of packages and features that framework core developers need to maintain.
- To reduce the number of packages that are downloaded and autoloaded.
- To remove a feature that is not essential.
- The HTML package was removed from the core of Laravel 5 and is now an external package. In this case, any of the previous reasons could be cited for the reason that this package was removed.
- HTML, which helps developers build forms, can be used if the frontend developer is also a backend or full-stack developer and prefers Laravel's way of doing things. In other situations, however, the web application HTML interface can be built using a JavaScript framework or a library, such as AngularJS or Backbone.js. In this case, the Laravel form package would not be necessary. Alternatively, as previously stated, Laravel can be used to

create an application that is merely a RESTful API. In this case, including the HTML package in the framework core would not be necessary and thus remains auxiliary.

In this particular case, certain Laravel packages were removed to lighten up the overall experience and to move toward a more *component-based* approach, which is similar to that used in Symfony.

Installing the HTML package

If you desire to use the HTML package in Laravel 5, installing it is a simple process. A group of developers in the Laravel community formed a repository called the Laravel collective, where the packages that have been removed from Laravel are maintained. To install the HTML package, simple use the `composer` command to add the package to the application as follows:

```
$ composer require laravelcollective/html
```

Note

Note that the `illuminate/HTML` package has been deprecated.

This will install the HTML package and the `composer.json` will show you the package added to the `require` section as follows:

```
"require": {
    "laravel/framework": "5.0.*",
    "laravelcollective/html": "~5.0",
  },
```

At this point, the package is installed.

Now, we need to add the `HTMLServiceProvider` to the list of providers in the `config/app.php` file:

```
  'providers' => [
  ...
    'Collective\Html\HtmlServiceProvider',
  ...
  ],
```

Lastly, the `Form` and `Html` aliases need to be added to the `config/app.php` file, as shown here:

```
'aliases' => [
   ...
        'Form' => 'Collective\Html\FormFacade',
        'Html' => 'Collective\Html\HtmlFacade',
  ...
  ],
```

Building web pages with Laravel

Laravel's approach to building web content is flexible. As much or as little of Laravel can be used to create HTML. Laravel uses the `filename.blade.php` convention to state that the file should be parsed by the blade parser, which actually converts the file into plain PHP. The name blade was inspired by the .NET's razor templating engine, so this may be familiar to someone who has used it. Laravel 5 provides a working demonstration of a form in the `/resources/views/` directory. This view is shown when the `/home` route is requested and the user is not currently logged in. This form is obviously not created using the Laravel form methods.

The route is defined in the `routes` file as follows:

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

An explanation of how this route uses middleware to check how to perform the user authentication will be discussed in Chapter 7, *Filtering Requests with Middleware*.

The master template

This is the following `app` (or `master`) template:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1">
    <title>Laravel</title>

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

    <!-- Fonts -->
    <link href='//fonts.googleapis.com/css?family=Roboto:400,300'
rel='stylesheet' type='text/css'>

    <!-- HTML5 shim and Respond.js for IE8 support of HTML5
elements and media queries -->
    <!-- WARNING: Respond.js doesn't work if you view the page
via file:// -->
    <!--[if lt IE 9]>
        <script
src="https://oss.maxcdn.com/html5shiv/3.7.2/html5shiv.min.js">
</script>
        <script
src="https://oss.maxcdn.com/respond/1.4.2/respond.min.js">
</script>
    <![endif]-->
</head>
<body>
    <nav class="navbarnavbar-default">
        <div class="container-fluid">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle
collapsed" data-toggle="collapse" data-target="#bs-example-
navbar-collapse-1">
                    <span class="sr-only">Toggle
Navigation</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <a class="navbar-brand" href="#">Laravel</a>
            </div>
```

```
            <div class="collapse navbar-collapse" id="bs-example-
navbar-collapse-1">
                <ul class="navnavbar-nav">
                    <li><a href="/">Home</a></li>
                </ul>

                <ul class="navnavbar-navnavbar-right">
                    @if (Auth::guest())
                        <li><a href="{{ route('auth.login')
}}">Login</a></li>
                        <li><a href="/auth/register">Register</a>
</li>
                    @else
                        <li class="dropdown">
                            <a href="#" class="dropdown-toggle"
data-toggle="dropdown" role="button" aria-expanded="false">{{
Auth::user()->name }} <span class="caret"></span></a>
                            <ul class="dropdown-menu"
role="menu">
                                <li><a
href="/auth/logout">Logout</a></li>
                            </ul>
                        </li>
                    @endif
                </ul>
            </div>
        </div>
    </nav>

    @yield('content')

    <!-- Scripts -->
    <script
src="//cdnjs.cloudflare.com/ajax/libs/jquery/2.1.3/jquery.min.js"
></script>
    <script src="//cdnjs.cloudflare.com/ajax/libs/twitter-
bootstrap/3.3.1/js/bootstrap.min.js"></script>
</body>
</html>
```

The Laravel 5 master template is a standard HTML5 template with the following features:

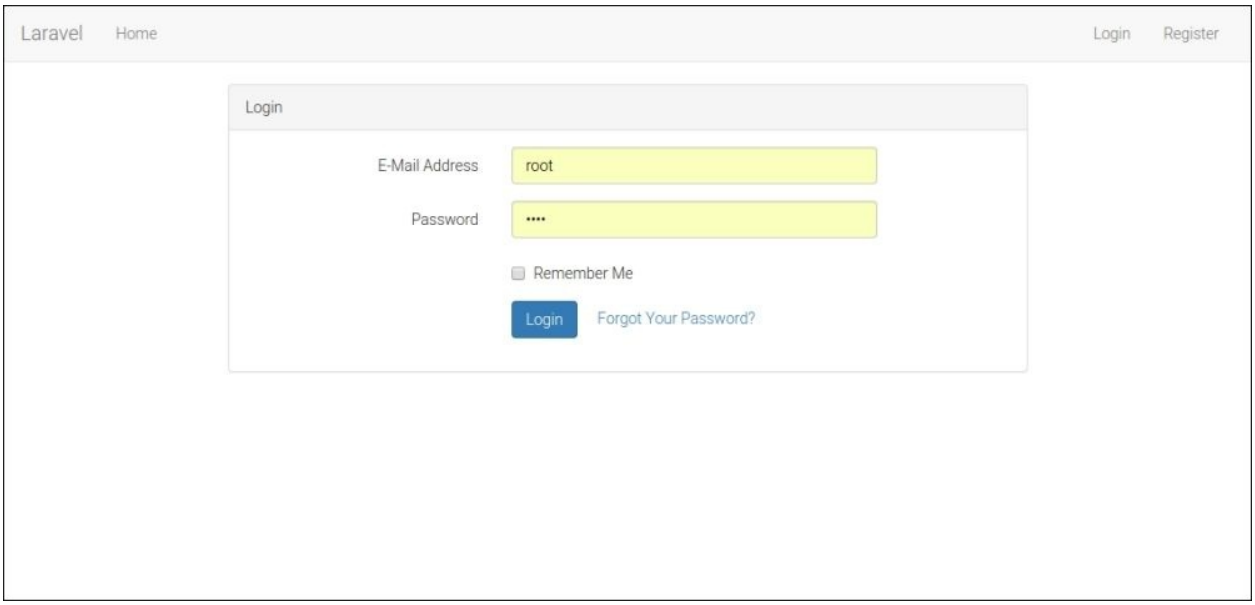
- If the browser is older than Internet Explorer 9:
  - Uses the HTML5 Shim from the CDN
  - Uses the Respond.js JavaScript code from the CDN to retrofit media queries and CSS3 features
- Using `@if (Auth::guest())`, if the user is not authenticated, the login form

is displayed; otherwise, the logout option is displayed

- Twitter bootstrap 3.x is included in the CDN
- The jQuery2.x is included in the CDN
- Any template that extends this template can override the content section

An example page

The following screenshot shows you the login page:



The source code for the login page is as follows:

```
@extends('app')
@section('content')
<div class="container-fluid">
    <div class="row">
        <div class="col-md-8 col-md-offset-2">
            <div class="panel panel-default">
                <div class="panel-heading">Login</div>
                <div class="panel-body">
                    @if (count($errors) > 0)
                        <div class="alert alert-danger">
                            <strong>Whoops!</strong> There were
some problems with your input.<br><br>
                            <ul>
                                @foreach ($errors->all() as
$error)
                                    <li>{{ $error }}</li>
                                @endforeach
                            </ul>
                        </div>
                    @endif
```

```
                    <form class="form-horizontal" role="form"
method="POST" action="/auth/login">
                        <input type="hidden" name="_token"
value="{{ csrf_token() }}">

                        <div class="form-group">
                            <label class="col-md-4 control-
label">E-Mail Address</label>
                            <div class="col-md-6">
                                <input type="email" class="form-
control" name="email" value="{{ old('email') }}">
                            </div>
                        </div>

                        <div class="form-group">
                            <label class="col-md-4 control-
label">Password</label>
                            <div class="col-md-6">
                                <input type="password"
class="form-control" name="password">
                            </div>
                        </div>

                        <div class="form-group">
                            <div class="col-md-6 col-md-offset-
4">
                                <div class="checkbox">
                                    <label>
                                        <input type="checkbox"
name="remember"> Remember Me
                                    </label>
                                </div>
                            </div>
                        </div>

                        <div class="form-group">
                            <div class="col-md-6 col-md-offset-
4">
                                <button type="submit" lass="btn
btn-primary" style="margin-right: 15px;">
                                    Login
                                </button>

                                <a href="/password/email">Forgot
Your Password?</a>
                            </div>
                        </div>
                    </form>
                </div>
            </div>
```

```
        </div>
    </div>
</div>
@endsection
```

From static HTML to static methods

This login page begins with the following:

```
@extends('app')
```

It obviously uses the object-oriented paradigm to state that the `app.blade.php` template will be rendered. The following line overrides the content:

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

For this exercise, the form builder will be used instead of the static HTML.

The form tag

We will convert a static `form` tag to a `FormBuilder` method. The HTML is as follows:

```
<form class="form-horizontal" role="form" method="POST"
action="/auth/login">
```

The method facade that we will use is as follows:

```
Form::open();
```

In the `FormBuilder.php` class, the `$reserved` attribute is defined as follows:

```
protected $reserved = ['method', 'url', 'route', 'action',
'files'];
```

The attributes that we need to pass to an array to the `open()` method are class, role, method, and action. Since method and action are reserved words, it is necessary to pass the parameters in the following manner:

Laravel form facade method array element	**HTML Form tag attribute**
method	method
url	action

role	role
class	class

Thus, the method call looks like this:

```
{!!
  Form::open(['class'=>'form-horizontal',
  'role =>'form',
  'method'=>'POST',
  'url'=>'/auth/login'])
!!}
```

The `{!! !!}` tags are used to start and end parsing of the form builder methods. The form method, `POST`, is placed first in the list of attributes in the HTML form tag.

Tip

The `action` attribute actually needs to be a `url`. If the `action` parameter is used, then it refers to the controller action. In this case, the `url` parameter produces the `action` attribute of the `form` tag.

Other attributes will be passed to the array and added to the list of attributes. The resultant HTML will be produced as follows:

```
<form method="POST" action="http://laravel.example/auth/login"
accept-charset="UTF-8" class="form-horizontal" role="form">

<input name="_token" type="hidden"
value="wUY2hFSEWCzKHFfhywHvFbq9TXymUDiRUFreJD4h">
```

The CRSF token is automatically added, as the `form` method is `POST`.

The text input field

To convert the input fields, a facade is used. The input field's HTML is as follows:

```
<input type="email" class="form-control" name="email" value="{{
old('email') }}">
```

Converting the preceding input field using a façade looks like this:

```
{!! Form::input('email','email',old('email'),['class'=>'form-
control' ]) !!}
```

Similarly, the text field becomes:

```
{!! Form::input('password','password',null,['class'=>'form-
control']) !!}
```

The input fields have the same signature. Of course, this can be refactored as follows:

```
<?php $inputAttributes = ['class'=>'form-control'] ?>
{!! Form::input('email','email',old('email'),$inputAttributes )
!!}
...
{!! Form::input('password','password',null,$inputAttributes ) !!}
```

The label tag

The `label` tags are as follows:

```
<label class="col-md-4 control-label">E-Mail Address</label>
<label class="col-md-4 control-label">Password</label>
```

To convert the `label` tags (`E-Mail Address` and `Password`), we will first create an array to hold the attributes, and then pass this array to the labels, as follows:

```
$labelAttributes = ['class'=>'col-md-4 control-label'];
```

Here is the form label code:

```
{!! Form::label('email', 'E-Mail Address', $labelAttributes) !!}
{!! Form::label('password', 'Password', $labelAttributes) !!}
```

Checkbox

To convert the checkbox to a facade, we will convert this:

```
<input type="checkbox" name="remember"> Remember Me
```

The preceding code is converted to the following code:

```
{!! Form::checkbox('remember','') !!} Remember Me
```

Tip

Remember that the PHP parameters should be sent in single quotation marks if there are no variables or other special characters, such as line breaks, inside the string to parse, while the HTML produced will have double quotes.

The submit button

Lastly, the submit button will be converted as follows:

```
<button type="submit" class="btn btn-primary" style="margin-
right: 15px;">
    Login
</button>
```

The preceding code after conversion is as follows:

```
    {!!
        Form::submit('Login',
        ['class'=>'btn btn-primary',
        'style'=>'margin-right: 15px;'])
     !!}
```

Tip

Note that the array parameter provides an easy way to provide any desired attributes, even those that are not among the list of standard HTML form elements.

The anchor tag with links

To convert the links, a helper method is used. Consider the following line of code:

```
<a href="/password/email">Forgot Your Password?</a>
```

The preceding line of code after conversion becomes:

```
{!! link_to('/password/email', $title = 'Forgot Your Password?',
$attributes = array(), $secure = null) !!}
```

Note

The `link_to_route()` method may be used to link to a route. For similar helper functions, visit http://laravelcollective.com/docs/5.0/html.

Closing the form

To end the form, we'll convert the traditional HTML form tag `</form>` to a

Laravel `{!! Form::close() !!}` form method.

The resultant form

By putting everything together, the page now looks like this:

```
@extends('app')
@section('content')
<div class="container-fluid">
  <div class="row">
    <div class="col-md-8 col-md-offset-2">
      <div class="panel panel-default">
        <div class="panel-heading">Login</div>
          <div class="panel-body">
            @if (count($errors) > 0)
                <div class="alert alert-danger">
                    <strong>Whoops!</strong> There were some
problems with your input.<br><br>
                    <ul>
                        @foreach ($errors->all() as $error)
                            <li>{{ $error }}</li>
                        @endforeach
                    </ul>
                </div>
            @endif
            <?php $inputAttributes = ['class'=>'form-control'];
                $labelAttributes = ['class'=>'col-md-4 control-
label']; ?>
            {!! Form::open(['class'=>'form-
horizontal','role'=>'form','method'=>'POST','url'=>'/auth/login']
) !!}
                <div class="form-group">
                    {!! Form::label('email', 'E-Mail
Address',$labelAttributes) !!}
                    <div class="col-md-6">
                    {!! Form::input('email','email',old('email'),
$inputAttributes) !!}
                    </div>
                </div>
                <div class="form-group">
                    {!! Form::label('password',
'Password',$labelAttributes) !!}
                    <div class="col-md-6">
                        {!!
Form::input('password','password',null,$inputAttributes) !!}
                    </div>
                </div>
                <div class="form-group">
                    <div class="col-md-6 col-md-offset-4">
```

```
                        <div class="checkbox">
                          <label>
                             {!! Form::checkbox('remember','')
!!} Remember Me
                          </label>
                        </div>
                    </div>
                </div>
                <div class="form-group">
                    <div class="col-md-6 col-md-offset-4">
                        {!! Form::submit('Login',['class'=>'btn
btn-primary', 'style'=>'margin-right: 15px;']) !!}
                        {!! link_to('/password/email', $title =
'Forgot Your Password?', $attributes = array(), $secure = null);
!!}
                    </div>
                </div>
            {!! Form::close() !!}
          </div>
      </div>
    </div>
  </div>
</div>
@endsection
```

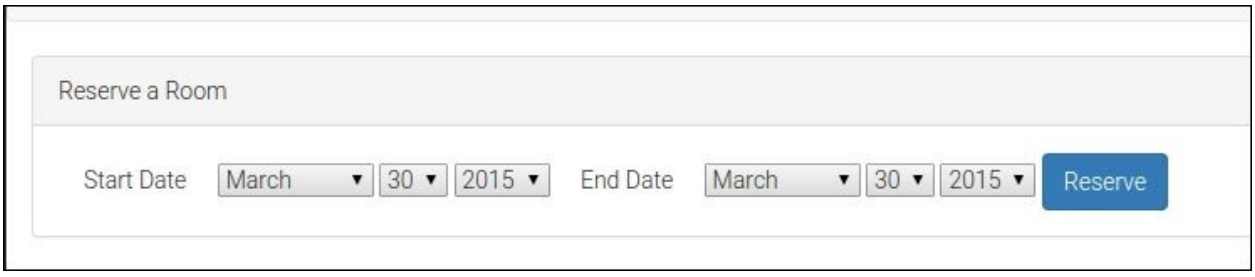
Our example

If we want to create a form to reserve a room in our accommodation, we can easily call a route from our controller:

```
/**
 * Show the form for creating a new resource.
 *
 * @return Response
 */
public function create()
{
    return view('auth/reserve');
}
```

Now we need to create a new view that is located at `resources/views/auth/reserve.blade.php`.

In this view, we can create a form to reserve a room in an accommodation where the user can select the start date, which comprises of the start day of the month and year, and the end date, which also comprises of the start day of the month and year:



The form would begin as before, with a POST to `reserve-room`. Then, the form label would be placed next to the select input fields. Finally, the day, the month, and the year select form elements would be created as follows:

```
{!! Form::open(['class'=>'form-horizontal',
        'role'=>'form',
        'method'=>'POST',
        'url'=>'reserve-room']) !!}
        {!! Form::label(null, 'Start Date',$labelAttributes) !!}
```

```
        {!! Form::selectMonth('month',date('m')) !!}
        {!! Form::selectRange('date',1,31,date('d')) !!}
        {!! Form::selectRange('year',date('Y'),date('Y')+3) !!}

        {!! Form::label(null, 'End Date',$labelAttributes) !!}

        {!! Form::selectMonth('month',date('m')) !!}
        {!! Form::selectRange('date',1,31,date('d')) !!}
        {!!
Form::selectRange('year',date('Y'),date('Y')+3,date('Y')) !!}

        {!! Form::submit('Reserve',
        ['class'=>'btn btn-primary',
        'style'=>'margin-right: 15px;']) !!}
{!! Form::close() !!}
```

Month select

Firstly, in the `selectMonth` method, the first parameter is the name of the input attribute, while the second attribute is the default value. Here, the PHP date method is used to extract the numeric portion of the current month—March in this case:

```
{!! Form::selectMonth('month',date('m')) !!}
```

The output, shown here formatted, is as follows:

```
<select name="month">
    <option value="1">January</option>
    <option value="2">February</option>
    <option value="3" selected="selected">March</option>
    <option value="4">April</option>
    <option value="5">May</option>
    <option value="6">June</option>
    <option value="7">July</option>
    <option value="8">August</option>
    <option value="9">September</option>
    <option value="10">October</option>
    <option value="11">November</option>
    <option value="12">December</option>
</select>
```

Date select

A similar technique is applied for the selection of the date, but using the `selectRange` method, the range of the days in the month are passed to the method. Similarly, the PHP date function is used to send the current date to the method as the fourth parameter:

```
{!! Form::selectRange('date',1,31,date('d')) !!}
```

Here is the formatted output:

```
<select name="date">
    <option value="1">1</option>
    <option value="2">2</option>
    <option value="3">3</option>
    <option value="4">4</option>
    ...
    <option value="28">28</option>
    <option value="29">29</option>
    <option value="30" selected="selected">30</option>
    <option value="31">31</option>
</select>
```

The date that should be selected is 30, since today is March 30, 2015.

Tip

For the months that do not have 31 days, usually a JavaScript method would be used to modify the number of days based on the month and/or the year.

Year select

The same technique that is used for the date range is applied for the selection of the year; once again, using the `selectRange` method. The range of the years is passed to the method. The PHP date function is used to send the current year to the method as the fourth parameter:

```
{!! Form::selectRange('year',date('Y'),date('Y')+3,date('Y')) !!}
```

Here is the formatted output:

```
<select name="year">
    <option value="2015" selected="selected">2015</option>
    <option value="2016">2016</option>
    <option value="2017">2017</option>
    <option value="2018">2018</option>
</select>
```

Here, the current year that is selected is 2015.

Form macros

We have the same code that generates our month, date, and year selection form block two times: once for the start date and once for the end date. To refactor the code, we can apply the DRY (don't repeat yourself) principle and create a form macro. This will allow us to avoid calling the form element creation method twice, as follows:

```
<?php
Form::macro('monthDayYear',function($suffix='')
{
    echo Form::selectMonth(($suffix!=='')?'month-
'.$suffix:'month',date('m'));
    echo Form::selectRange(($suffix!=='')?'date-
'.$suffix:'date',1,31,date('d'));
    echo Form::selectRange(($suffix!=='')?'year-
'.$suffix:'year',date('Y'),date('Y')+3,date('Y'));
});
?>
```

Here, the month, date, and year generation code is placed into a macro, which is inside the PHP tags, and it is necessary to add `echo` to print out the result. The `monthDayYear` name is given to this macro method. Calling our macro two times: once after each label; each time adding a different suffix via the `$suffix` variable. Now, our form code looks like this:

```
<?php
Form::macro('monthDayYear',function($suffix='')
{
    echo Form::selectMonth(($suffix!=='')?'month-
'.$suffix:'month',date('m'));
    echo Form::selectRange(($suffix!=='')?'date-
'.$suffix:'date',1,31,date('d'));
    echo Form::selectRange(($suffix!=='')?'year-
'.$suffix:'year',date('Y'),date('Y')+3,date('Y'));
});
?>
{!! Form::open(['class'=>'form-horizontal',
                'role'=>'form',
                'method'=>'POST',
                'url'=>'/reserve-room']) !!}
    {!! Form::label(null, 'Start Date',$labelAttributes) !!}
    {!! Form::monthDayYear('-start') !!}
    {!! Form::label(null, 'End Date',$labelAttributes) !!}
    {!! Form::monthDayYear('-end') !!}
    {!! Form::submit('Reserve',['class'=>'btn btn-primary',
```

```
            'style'=>'margin-right: 15px;']) !!}
{!! Form::close() !!}
```

Conclusion

The choice to include the HTML form generation package in Laravel 5 can ease the burden of having to create numerous HTML forms. This approach allows developers to use methods, create reusable macros, and use a familiar Laravel approach to build the frontend. Once the basic methods are learned, it is very easy to simply copy and paste the previously created form elements, and then change their element's name and/or the array that is sent to them.

Depending on the size of the project, this approach may or may not be the right choice. For a very small application, the difference in the amount of code that needs to be written is not very evident, although, as is the case with the `selectMonth` and `selectRange` methods, the amount of code necessary is drastic.

This technique, combined with the use of macros, makes it easy to reduce the occurrence of copy duplication. Also, one of the major problems with the frontend design is that the contents of the class of the various elements may need to change throughout the entire application. This would mean performing a large find and replace operation, where changes are required to be made to HTML, such as changing class attributes. By creating an array of attributes, including class, for similar elements, changes made to the entire form can be performed simply by modifying the array that those elements use.

In a larger project, however, where parts of forms may be repeated throughout the application, the wise use of macros can easily reduce the amount of code necessary to be written. Not only this, but macros can isolate the code inside from changes that would require more than one block of code to be changed throughout multiple files. In the example, where the month, date, and year is to be selected, it is possible that this could be used up to 20 times in a large application. Any changes made to the desired block of HTML can be simply done to the macro and the result would be reflected in all of the elements that use it.

Ultimately, the choice of whether or not to use this package will reside with the developer and the designer. Since a designer who wants to use an alternative frontend design tool may not prefer, nor be competent, to work with the methods in the package, he or she may want to not use it.

Summary

In this chapter, the history and the installation of the HTML Laravel composer package was outlined. The construction of the master template was explained and then the form components, such as the various form input types, were shown through examples.

Finally, the construction of a form for the room reservation, as used in the book's example software, was explained, as well as a "do not repeat yourself" form macro creation technique.

In the next chapter, we will take a look at a way to use annotations to reduce the time required to create routing for the application's controllers.

Chapter 6. Taming Complexity with Annotations

In the previous chapter, you learned how to create a RESTful API that involved receiving a request from the Internet, routing it to the controllers, and processing it. In this chapter, you will learn how to use annotations in the DocBlock, a way of performing routing that requires even less code and can be a faster and more organized way of doing collaborative programming with a team.

Annotations will be demonstrated for:

- Routing of HTTP requests such as GET, POST, and PUT
- Turning a controller into a fully enabled CRUDL resource
- Listening to events that are fired from commands
- Adding middleware to controllers to limit or filter requests

Annotations are great mechanisms used in programming. Annotations are metadata that enhance other data. Since this may seem slightly confusing, so we need to first understand what the meaning of metadata is. The word **metadata** is a word that contains two parts:

- **meta**: This is a Greek word that means transcending or encompassing.
- **data**: This is a Latin word that means pieces of information.

Thus, metadata serves to enhance or extend the meaning of something.

Annotations in other programming languages

Next, we will discuss annotations that are used in computer programming. We will take a look at several examples from Java, C#, and PHP, and then finally, take a look at how annotations are used in Laravel.

Annotations in Java

Annotations were first proposed in Java Version 1.1 and added in Version 1.2. The following is an example of an annotation that is used to override an animal's `speak` method:

```
Java 1.2
/**
 * @author      Jon Doe <jon@doe.com>
 * @version     1.6                (current version number)
 * @since       2010-03-31         (version package...)
 */
public void speak() {
}

public class Animal {
    public void speak() {
    }
}
public class Cat extends Animal {
    @Override
    public void speak() {
        System.out.println("Meow.");
    }
 }
```

Note that the `@` symbol is used to signal the compiler that this annotation, `@Override`, is important.

Annotations in C#

In C#, annotations are called attributes and use square brackets instead of the more often used `@` symbol:

```
[AttributeUsageAttribute(AttributeTargets.Property|AttributeTarge
ts.Field, AllowMultiple = false, Inherited = true)]
public sealed class AssociationAttribute : Attribute
```

Annotations in PHP

Other PHP frameworks use annotations. Symfony makes extensive use of annotations. In **Doctrine**, which is Symfony's ORM and is similar to Laravel's Eloquent, annotations are used to define relationships. Symfony also uses annotations for routing. The **Zend Framework** (**ZF**) uses annotations as well. Both the testing tools Behat and PHPUnit use annotations. In the following example of Behat, an annotation is used to indicate that this method should be executed before the test suite:

```
/**
 * @BeforeSuite
 */
public static function prepare(SuiteEvent $event)
{
// prepare system for test suite
// before it runs
}
```

DocBlock annotations

The example of annotation usage shown in the previous Behat example is rather interesting, as it places the annotation inside the DocBlock. The DocBlock begins with a forward slash and two asterisks:

```
/**
```

It contains *n* lines beginning with an asterisk.

The DocBlock ends with a single asterisk and forward slash:

```
 */
```

This syntax tells the parser that something useful is inside the DocBlock in addition to the normal comments.

DocBlock annotations in Laravel

When Laravel 5 was being developed, support for routing and event listeners via DocBlock annotations was originally added. Its syntax was similar to Symfony and Zend.

Symfony

The syntax for Symfony is as follows:

```
/**
 * @Route("/accommodations/search")
 * @Method({"GET"})
 */

public function searchAction($id)
{
```

Zend

The syntax for Zend is as follows:

```
/**
 * @Route(route="/accommodations/search")
 */

public function searchAction()
{
```

Laravel

The syntax for Laravel is as follows:

```
/**
 * @Get("/hotels/search")
 */

public function search()
{
```

What type of problem does the DocBlock annotation try to solve though?

One use of doc-annotations would be to add them to controllers, thus moving the control of the routing and middleware to the controller. This would make the controller more portable and even framework-agnostic, since the `routes.php` file

would play a lesser role, if not be totally absent. As shown in the following example, the `routes.php` file can grow quite large, and this would lead to complexity and even cause the file to be unmanageable:

```
Route::patch('hotel/{hid}/room/{rid}','AccommodationsController@e
ditRoom');
Route::post('hotel/{hid}/room/{rid}','AccommodationsController@re
serve');
Route::get('hotel/stats,HotelController@Stats');
Route::resource('country', 'CountryController');
Route::resource(city', 'CityController');
Route::resource('state', 'StateController');
Route::resource('amenity', 'AmenitiyController');
Route::resource('country', 'CountryController');
Route::resource(city', 'CityController');
Route::resource('country', 'CountryController');
Route::resource('city', 'CityController');
Route::resource('horse', 'HorseController');
Route::resource('cow', 'CowController');
Route::resource('zebra', 'ZebraController');
Route::get('dragon/{id}', 'DragonController@show');
Route::resource('giraffe', 'GiraffeController');
Route::resource('zebrafish', 'ZebrafishController');
```

The idea of DocBlock annotations would be to tame this complexity, as the routing would be moved to the controllers.

Shortly before the release of Laravel 5.0, due to community disapproval, the feature was removed. Also, since some developers may not want to use this approach, it would be proper to move this package from the core of Laravel and into a package. The method to install the package is similar to the way in which the HTML package was added. This package is also supported by the Laravel Collective. It is easy to add annotations by typing the following composer command:

```
$ composer require laravelcollective/annotations
```

This will install the annotations package, while `composer.json` will show the package added in the require section, as follows:

```
"require": {
    "laravel/framework": "5.0.*",
    "laravelcollective/annotations": "~5.0",
  },
```

The next step would be to create a file named `AnnotationsServiceProvider.php` and add the following code:

```
<?php namespace App\Providers;

use Collective\Annotations\AnnotationsServiceProvider as
ServiceProvider;

class AnnotationsServiceProvider extends ServiceProvider {

    /**
     * The classes to scan for event annotations.
     *
     * @var array
     */
    protected $scanEvents = [];

    /**
     * The classes to scan for route annotations.
     *
     * @var array
     */
    protected $scanRoutes = [];

    /**
     * The classes to scan for model annotations.
     *
     * @var array
     */
    protected $scanModels = [];

    /**
     * Determines if we will auto-scan in the local environment.
     *
     * @var bool
     */
    protected $scanWhenLocal = false;

    /**
     * Determines whether or not to automatically scan the
controllers
     * directory (App\Http\Controllers) for routes
     *
     * @var bool
     */
    protected $scanControllers = false;

    /**
     * Determines whether or not to automatically scan all
```

```
namespaced
     * classes for event, route, and model annotations.
     *
     * @var bool
     */
    protected $scanEverything = false;

}
```

Next, the `AnnotationsServiceProvider.php` file will need to be added to the `config/app.php` file. The class, which needs to be added with the namespace, should be added to the providers array as follows:

```
'providers' => [
    // ...
    'App\Providers\AnnotationsServiceProvider'
  ];
```

Resource controller using DocBlock annotations

Now, to illustrate the use of Laravel's DocBlock annotations, we will examine the steps.

First, we will create the accommodations controller as usual:

```
$ php artisan make:controller AccommodationsController
```

Next, we will add the accommodations controller to the list of the annotation service provider's routes to scan:

```
protected $scanRoutes = [
    'App\Http\Controllers\HomeController',
    'App\Http\Controllers\AccommodationsController'
];
```

Now, we will add the DocBlock annotation to the controller. In this case, we will instruct the parser to use this controller as a resource controller for the accommodations route. The code to be added is as follows:

```
/**
* @Resource("/accommodations")
*/
```

Since the whole controller will be turned into a resource, the DocBlock annotation should be inserted before the class definition. The `AccommodationsController` class should now be as follows:

```
<?php namespace MyCompany\Http\Controllers;

use Illuminate\Support\Facades\Response;
use MyCompany\Http\Requests;
use MyCompany\Http\Controllers\Controller;
use MyCompany\Accommodation;
use Illuminate\Http\Request;

/**
* @Resource("/accommodations")
*/
class AccommodationsController extends Controller {
```

```
    /**
     * Display a listing of the resource.
     *
     * @return Response
     */
    public function index(Accommodation $accommodation)
    {
        return $accommodation->paginate();
    }
```

Note

Note that double quotes are required here:

```
@Resource("/accommodations")
```

The following syntax, using single quotes, would not be correct and will not function:

```
@Resource('/accommodations')
```

Single method routing

If we desire to just add one route to a single method, such as "search for accommodation", then an annotation would be added above the single method; however, this time, inside the class. To handle the GET HTTP request verb, the code would be as follows:

```
/**
 * Search for an accommodation
 * @Get("/search-accommodation")
 */
```

The class would be as follows:

```
<?php namespace MyCompany\Http\Controllers;

use Illuminate\Support\Facades\Response;
use MyCompany\Http\Requests;
use MyCompany\Http\Controllers\Controller;
use MyCompany\Accommodation;
use Illuminate\Http\Request;

class AccommodationsController extends Controller {

    /**
    * Search for an accommodation
    * @Get("/search-accommodation")
    */
    public function index(Accommodation $accommodation)
    {
        return $accommodation->paginate();
    }
```

Scanning routes

The next step is extremely important. The Laravel application must process the annotations. For this chore, Artisan is used to scan the routes.

The following command is used to scan the routes. The output will be `Routes scanned!` as shown here:

```
$ php artisan route:scan

Routes scanned!
```

The results of this scan will produce a file named `routes.scanned.php` in the `storage/framework` directory.

The following code writes to `storage/framework/routes.scanned.php` file:

```
$router->get('search-accommodation', [
  'uses' =>
'MyCompany\Http\Controllers\AccommodationsController@search',
  'as' => NULL,
  'middleware' => [],
  'where' => [],
  'domain' => NULL,
]);
```

Note

Note that the `storage/framework/routes.scanned.php` file does not need to be put into source code control as it is generated.

Automatic scanning

If a developer has to execute the Artisan route scan command as the controllers are being built, the effort to do this could become tedious. As a convenience to the developer, there is way to have Laravel automatically scan the controllers in the `scanRoutes` array on every request to the framework while in the development mode.

In the `AnnotationsServiceProvider.php` file, set the `scanWhenLocal` attribute to `true`.

The same is true for `$scanControllers` and `$scanEverything`; these two Boolean flags allow the framework to automatically scan the `App\Http\Controllers` directory, and any class that is namespaced, respectively.

It is imperative to remember that this should *only* be used during development and on a development machine, since it will add unnecessary overhead to the request cycle. An example of how the attributes are set to `true` is shown in the following code:

```
<?php namespace App\Providers;

use Collective\Annotations\AnnotationsServiceProvider as
ServiceProvider;

class AnnotationsServiceProvider extends ServiceProvider {

    /**
     * The classes to scan for event annotations.
     *
     * @var array
     */
    protected $scanEvents = [];

    …

    /**
     * Determines if we will auto-scan in the local environment.
     *
     * @var bool
     */
    protected $scanWhenLocal = true;
```

```
    /**
     * Determines whether or not to automatically scan the
controllers
     * directory (App\Http\Controllers) for routes
     *
     * @var bool
     */
    protected $scanControllers = true;

    /**
     * Determines whether or not to automatically scan all
namespaced
     * classes for event, route, and model annotations.
     *
     * @var bool
     */
    protected $scanEverything = true;

}
```

Enabling these options will slow down the framework, but allow flexibility in the development phase.

Additional annotations

To pass an ID to a route, as is common during the display of a single accommodation, the code would be as follows:

```
/**
* Display the specified resource.
* @Get("/accommodation/{id}")
*/
```

This DocBlock annotation would be placed above the function inside the class, which is similar to the previous example.

To limit the ID to one or more digits, an `@Where` annotation can be used as follows:

```
@Where({"id": "\d+"})
```

Both the annotations are combined together as shown in the following code:

```
/**
 * Display the specified resource.
 * @Get("/accommodation/{id}")
 * @Where({"id": "\d+"})
 */
```

To add middleware to the example, which limits the request to only authenticated users, the `@Middleware` annotation may be used:

```
/**
 * Display the specified resource.
 * @Get("/accommodation/{id}")
 * @Where({"id": "\d+"})
 * @Middleware("auth")
 */
```

HTTP verbs

The following is a list of the various HTTP verbs that can use annotations, that mirror RESTful standards:

- `@Delete`: This verb deletes a resource.
- `@Get`: This verb displays a resource or resources.
- `@Options`: This verb displays a list of options.
- `@Patch`: This verb modifies an attribute or attributes of a resource.
- `@Post`: This verb creates a new resource.
- `@Put`: This verb modifies a resource.

Other annotations

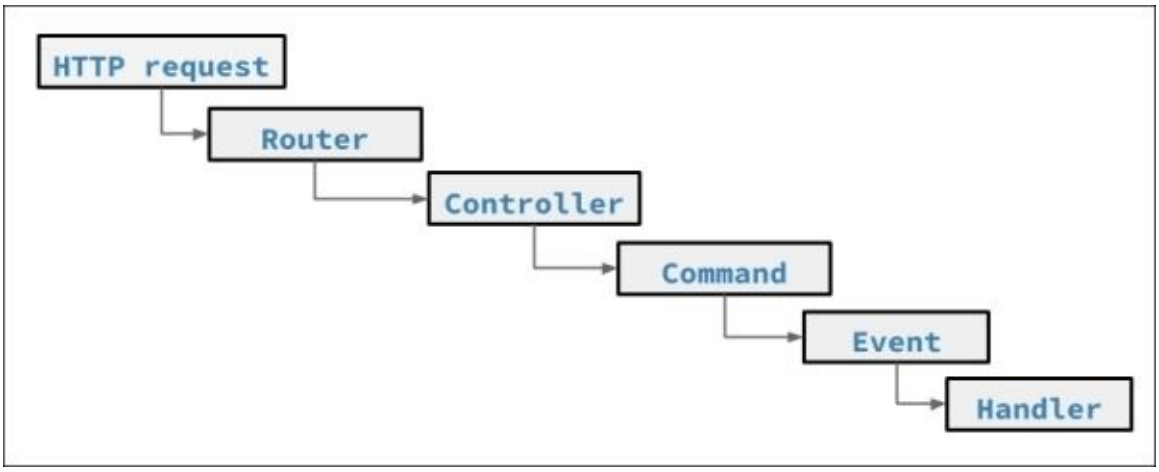
There are additional annotations that also may be used in controllers. The annotations are as follows:

- `@Any`: This responds to any HTTP request.
- `@Controller`: This creates a controller for a resource.
- `@Middleware`: This adds middleware to a resource.
- `@Route`: This enables a route.
- `@Where`: This limits requests based on certain criteria.
- `@Resource`: This enables a resource.

Using annotations in Laravel 5

Let us recall the pathway that was implemented in Laravel, as illustrated here:

- The HTTP request gets routed to a controller
- The command is instantiated inside of the controller
- An event is fired
- The event is handled



Laravel's modern command-based pub-sub pathway.

Using annotations, this process can be rendered even easier. First, a reservations controller will be created:

```
$ php artisan make:controller ReservationsController
```

To create a route to allow the user to create a new reservation, the POST HTTP verb will be used. The `@Post` annotation will listen to requests with the `POST` method attached to the `/bookRoom` url. This is used in place of the route that would normally be found inside of the `routes.php` file:

```
<?php namespace MyCompany\Http\Controllers;

use ...

class ReservationsController extends Controller {
/**
* @Post("/bookRoom")
```

```
*/
  public function reserve()
  {
  }
```

If we want to limit requests to a valid URL, the domain parameter limits requests to a certain URL. Additionally, the auth middleware requires the authentication of any request that wishes to reserve a room:

```
<?php namespace App\Http\Controllers;

use …
/**
* @Controller(domain="booking.hotelwebsite.com")
*/

class ReservationsController extends Controller {

/**
* @Post("/bookRoom")
* @Middleware("auth")
*/
  public function reserve()
  {
```

Next, the `ReserveRoom` command should be created. This command will be instantiated inside the controller:

```
$ php artisan make:command ReserveRoom
```

Contents of the ReserveRoom Command are as follows:

```
<?php namespace MyCompany\Commands;

use MyCompany\Commands\Command;
use MyCompany\User;
use MyCompany\Accommodation\Room;
use MyCompany\Events\RoomWasReserved;

use Illuminate\Contracts\Bus\SelfHandling;

class ReserveRoomCommand extends Command implements SelfHandling
{

  public function __construct()
  {
  }
  /**
```

```
   * Execute the command.
   */
  public function handle()
  {
  }
}
```

Next, we will need to instantiate the `ReserveRoom` command from inside the reservation controller:

```
<?php namespace MyCompany\Http\Controllers;

use MyCompany\Accommodation\Reservation;
use MyCompany\Commands\PlaceOnWaitingListCommand;
use MyCompany\Commands\ReserveRoomCommand;
use MyCompany\Events\RoomWasReserved;
use MyCompany\Http\Requests;
use MyCompany\Http\Controllers\Controller;
use MyCompany\User;
use MyCompany\Accommodation\Room;

use Illuminate\Http\Request;

class ReservationsController extends Controller {

/**
 * @Post("/bookRoom")
 * @Middleware("auth")
 */
  public function reserve()
  {
    $this->dispatch(
    new ReserveRoom(\Auth::user(),$start_date,$end_date,$rooms)
    );
  }
```

Now we will create the RoomWasReserved event:

```
$ php artisan make:event RoomWasReserved
```

To instantiate the `RoomWasReserved` event from the `ReserveRoom` handler, we may take advantage of the `event()` helper method. In this example, the command is self-handling, thus it is simple to do this:

```
<?php namespace App\Commands;

use App\Commands\Command;
use Illuminate\Contracts\Bus\SelfHandling;
```

```
class ReserveRoom extends Command implements SelfHandling {
    public function __construct(User $user, $start_date,
$end_date, $rooms)
    {
    }
    public function handle()
    {
        $reservation = Reservation::createNew();
        event(new RoomWasReserved($reservation));
    }
}
```

Since the user needs to be sent the details of the room reservation email, the next step is to create an e-mail sender handler for the `RoomWasReserved` event. To do this, `artisan` is used again to create the handler:

```
$ php artisan handler:event RoomReservedEmail –
event=RoomWasReserved
```

The methods of the `SendEmail` handler for the `RoomWasReserved` event are simply constructor and handler. The job of sending the e-mail would be performed inside the handler method. The `@Hears` annotation is added to its DocBlock to complete the process:

```
<?php namespace MyCompany\Handlers\Events;

use MyCompany\Events\RoomWasReserved;

use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldBeQueued;

class RoomReservedEmail {
  public function __construct()
  {
  }

  /**
   * Handle the event.
   * @Hears("\App\Events\RoomWasReserved")
   * @param  RoomWasReserved  $event
   */
  public function handle(RoomWasReserved $event)
  {
     //TODO: send email to $event->user
  }
}
```

Simply add `RoomReservedEmail` to the `scanEvents` array to allow that event to be scanned as follows:

```
protected $scanEvents = [
   'App\Handlers\Events\RoomReservedEmail'
];
```

The last step is to import. Artisan is used to scan events for annotations and write to the output file:

```
$ php artisan event:scan

  Events scanned!
```

Here is the output of the `storage/framework/events.scanned.php` file, which shows the event listener:

```
<?php $events->listen(array(0 => 'App\\Events\\RoomWasReserved',
), App\Handlers\Events\RoomReservedEmail@handle');
```

The final view of the scanned annotation files inside the storage directory is as follows. Note that they are parallel:

```
storage/framework/events.scanned.php
storage/framework/routes.scanned.php
```

Tip

Laravel uses `artisan` to cache routes, but not to scan events, so the following command produces a cached file:

```
$ php artisan route:cache
Route cache cleared!
Routes cached successfully!
```

The `route:scan` command must be run before the `route:cache` is run, so it is important to execute the two commands in this order:

```
$ php artisan route:scan
Routes scanned!

$ php artisan route:cache
Route cache cleared!
Routes cached successfully!
```

This command writes to: `storage/framework/routes.php`.

```
<?php

app('router')->setRoutes(

unserialize(base64_decode('TzozNDoiSWxsdW1pbmF0ZVxSb3V0aW5nXFd…')
)
);
```

Both the files will be created, but only the compiled `routes.php` file is used until `php artisan route:scan` is run again.

Advantages

There are several principle advantages of using DocBlock annotations for routing in Laravel:

- Each controller remains independent. The controller is not "tied" to a separate route and this makes sharing controllers, and moving them from project to project, easier. The `routes.php` file may be viewed as unnecessary for a simple project with only a few controllers.
- Developers need not worry about `routes.php`. When working with other developers, the routes file needs to be kept continually in synchronization. With the DocBlock annotations approach, the `routes.php` file is cached and not placed under source code control; each developer may simply concentrate on his or her controllers.
- The route annotation keeps the route together with the controller. When controllers and routes are separate, and when a new programmer reads the code for the first time, it may not be immediately clear to which routes each controller method is attached. By placing the route directly in the DocBlock above the function, it is immediately evident.
- Developers that are familiar with, and are used to, using annotations in frameworks, such as Symfony and Zend may find using annotations in Laravel to be a very convenient way of developing software applications. Also, Java and C# developers using Laravel as their first PHP experience will find annotations very convenient.

Conclusion

The decision about whether or not to use annotations in a software lies with the developer. The decision to remove it from the core of Laravel, as well as the HTML forms packages, shows that the framework is becoming more and more flexible, having as default only a minimum set of packages. This allows for stability and less maintenance for the core developers as Laravel moves towards a **long-term-support** (**LTS**) release with Laravel 5.1.

Since the annotations package is part of the Laravel Collective, support for this package will be managed by the team, which guarantees that the feature's usefulness will be extended and expanded with contributions through to the repository.

Also, the package could be extended to include a template that is automatically created with the route annotation that has the same name as the controller. This would save yet another step in the process of creating a controller and route—one of the most essential yet monotonous tasks in the software development process.

Summary

In this chapter, we have learned about annotations, how they are used in programming in general, how they are used in other frameworks, and finally how their use has been adopted into the Laravel annotation composer package. We have learned how to speed up the development process through the use of annotations and how to automate the scanning of annotations. In the next chapter we will learn about middleware, which is a mechanism used in the *middle* between the route and the application.

Chapter 7. Filtering Requests with Middleware

In this chapter, middleware will be discussed in detail and examples from the accommodation software will be provided. Middleware is a great mechanism to help separate a software application into separate layers. To illustrate this principle, middleware provides layers of protection around the innermost part of the application, which could be thought of as the kernel.

In Laravel 4, middleware was known as filters. These filters were used in routing to perform actions that came before the controller like authentication, where the user would be filtered based on certain criteria. Also, the filters could come after the controller.

In Laravel 5, the concept of middleware, which was already present but not prominent in Laravel 4, is now brought into the foreground into the actual request workflow and can be used in various ways. Think of it as a Russian doll, where each doll represents a layer in the application—having the correct credentials will allow us to enter deeper into the application.

The HTTP kernel

The file located at `app/Http/Kernel.php` is a file that manages the configuration of the kernel of the program. The basic structure is as follows:

```
<?php namespace App\Http;

use Illuminate\Foundation\Http\Kernel as HttpKernel;

class Kernel extends HttpKernel {

  /**
   * The application's global HTTP middleware stack.
   *
   * @var array
   */
  protected $middleware = [

'Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode',
    'Illuminate\Cookie\Middleware\EncryptCookies',
    'Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse',
    'Illuminate\Session\Middleware\StartSession',
    'Illuminate\View\Middleware\ShareErrorsFromSession',
    'Illuminate\Foundation\Http\Middleware\VerifyCsrfToken',
  ];

  /**
   * The application's route middleware.
   *
   * @var array
   */
  protected $routeMiddleware = [
    'auth' => 'App\Http\Middleware\Authenticate',
    'auth.basic' =>
'Illuminate\Auth\Middleware\AuthenticateWithBasicAuth',
    'guest' => 'App\Http\Middleware\RedirectIfAuthenticated',
  ];

}
```

The `$middleware` array is a list of middleware classes and their namespace, and it is executed at every request. The `$routeMiddleware` array is a key and value array that is created as list of *aliases* that can be used together with routes to filter requests.

The basic middleware structure

The routing middleware classes implement the `Middleware` interface:

```
<?php namespace Illuminate\Contracts\Routing;

use Closure;

interface Middleware {

  /**
   * Handle an incoming request.
   *
   * @param  \Illuminate\Http\Request  $request
   * @param  \Closure  $next
   * @return mixed
   */
  public function handle($request, Closure $next);

}
```

In any class that implements this base class, there must be a `handle` method that accepts the `$request` as well as a `Closure`.

The basic structure of a middleware is as follows:

```
<?php namespace Illuminate\Foundation\Http\Middleware;

use Closure;
use Illuminate\Contracts\Routing\Middleware;
use Illuminate\Contracts\Foundation\Application;
use Symfony\Component\HttpKernel\Exception\HttpException;

class CheckForMaintenanceMode implements Middleware {

  /**
   * The application implementation.
   *
   * @var \Illuminate\Contracts\Foundation\Application
   */
  protected $app;

  /**
   * Create a new filter instance.
   *
   * @param  \Illuminate\Contracts\Foundation\Application  $app
   * @return void
```

```
     */
    public function __construct(Application $app)
    {
        $this->app = $app;
    }

    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        if ($this->app->isDownForMaintenance())
        {
            throw new HttpException(503);
        }
        return $next($request);
    }
}
```

Here, the `CheckForMaintenanceMode` middleware does exactly as its name suggests: the `handle` method checks if the application is in application mode. The `isDownForMaintenance` method of the app is called and if it returns `true`, then a 503 HTTP exception will be returned and execution of the method will stop. Otherwise, the `$next` closure with the `$request` parameter is returned to the calling class.

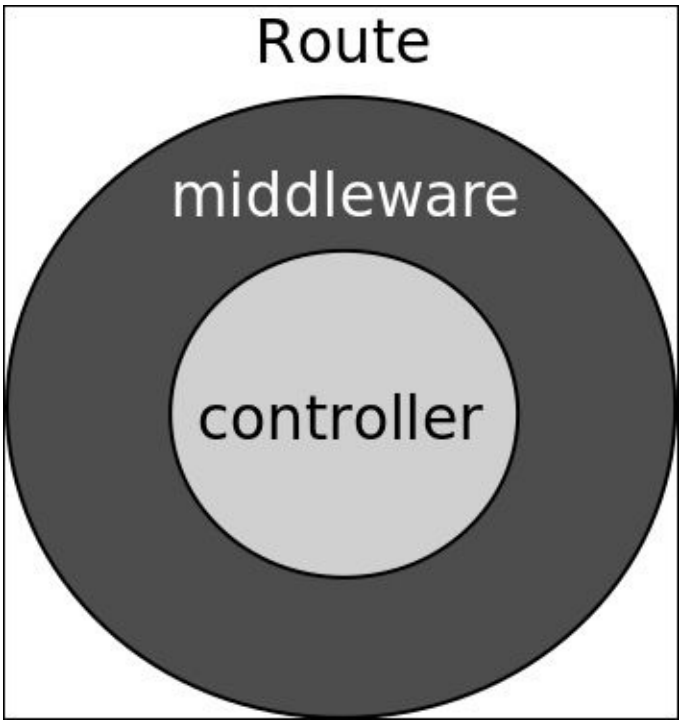
Tip

Middleware such as `CheckForMaintenanceMode` could be removed from the `$middleware` array and moved into the `$routeMiddleware` array to not require it to be executed upon every request, but only when desired from a certain route.

Route middleware unravelled

Two route-based middleware classes are present in Laravel 5 in `app/Http/Middleware/`. One of these classes is named `Authenticate`. It provides basic authentication and uses a contract.

In reference to routes, the middleware sits between the route and the controller:



Default middleware – the Authenticate class

A class called `Authenticate.php` has the following code:

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

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

class Authenticate {
  /**
   * The Guard implementation.
   *
   * @var Guard
   */
  protected $auth;

  /**
   * Create a new filter instance.
   *
   * @param  Guard  $auth
   * @return void
   */
  public function __construct(Guard $auth)
  {
    $this->auth = $auth;
  }

  /**
   * Handle an incoming request.
   *
   * @param  \Illuminate\Http\Request  $request
   * @param  \Closure  $next
   * @return mixed
   */
  public function handle($request, Closure $next)
  {
    if ($this->auth->guest())
    {
      if ($request->ajax())
      {
        return response('Unauthorized.', 401);
      }
      else
      {
        return redirect()->guest('auth/login');
      }
    }
```

```
    return $next($request);
  }
}
```

The first thing to note is `Illuminate\Contracts\Auth\Guard`, which handles the logic to check whether or not a user is logged in. It is injected into the constructor.

Contracts

Notice that the concept of a contract is a new way of using an interface to provide a nonconcrete class to separate the actual class from the calling class. This provides a nice layer of separation and allows the underlying class to be easily switched out if needed, while maintaining the parameters and return types of the methods.

Handle

The `handle` class is where the real work is done. The `$request` object is passed in along with the `$next` closure. What happens next is really simple yet important. The code asks if the current user is a guest, that is, not authenticated or logged in. If the user is not logged in, then the method will not allow the user to access the next step. If the request has arrived via Ajax, then a 401 message is returned to the browser.

If the request does not arrive via an Ajax request, the code assumes that the request has arrived via a standard page request and the user is directed to the auth/login page which allows the user to log in to the application. Otherwise, if the user is authenticated (`guest()` is not equal to `true`), then the `$next` closure is returned to the software application with the `$request` object as its parameter. To summarize, the execution of the application is stopped only if the user is not authenticated; otherwise, the execution continues.

The important thing to remember is that, in this case, the `$request` object is returned to the software.

Custom middleware – Log

It is simple to create custom middleware using Artisan. The `artisan` command is as follows:

```
$ php artisan make:middleware LogMiddleware
```

Our `LogMiddleware` class needs to be added to the `$middleware` array in the `Http/Kernel.php` file as shown here:

```
protected $middleware = [

'Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode',
  'Illuminate\Cookie\Middleware\EncryptCookies',
  'Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse',
  'Illuminate\Session\Middleware\StartSession',
  'Illuminate\View\Middleware\ShareErrorsFromSession',
  'MyCompany\Http\Middleware\LogMiddleware'
];
```

The `LogMiddleware` class is the name given to a middleware class that will be used to log users that use a website. The class produced has only one method, namely `handle`. As the authentication middleware, it accepts the `$request` object as well as the `$next` closure:

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

use Closure;

class LogMiddleware {

  /**
   * Handle an incoming request.
   *
   * @param  \Illuminate\Http\Request  $request
   * @param  \Closure  $next
   * @return mixed
   */
  public function handle($request, Closure $next)
  {
    return $next($request);
  }
}
```

In this case, we want to simply log both the user ID and the date and time at which some action was performed. The `$request` object is assigned to the `$response`

object and the `$response` object is returned instead of `$next`. The code is as follows:

```
public function handle($request, Closure $next)
{
  $response = $next($request);
  Log::create(['user_id'=>\Auth::user()-
>id,'created_at'=>date("Y-
  m-d H:i:s")]);
  return $response;
}
```

The Log model

Create the `Log` model using the following command:

```
$php artisan make:model Log
```

Set the `Log` model to use a table named `log` instead of `logs` by using the protected `$table` attribute. Next, set the model to not use a timestamp by setting the public `$timestamps` attribute to `false`. Finally, allow both the `user_id` and `created_at` fields to be populated at the same time by setting the protected `$fillable` attribute to an array of the fields that are desired to be filled, allowing use of the `create` function. The class will look like this after the preceding modifications:

```
<?php namespace MyCompany;

use Illuminate\Database\Eloquent\Model;

class Log extends Model {
    protected $table = 'log';
    public $timestamps = false;
    protected $fillable = ['user_id','created_at'];
}
```

We could also create the `Log` model as a polymorphic model, allowing it to be used in more than one context by added the following code to the `Log` model:

```
public function loggable()
{
     return $this->morphTo();
}
```

Tip

More information regarding this is available in the Laravel documentation.

Log model migration

It is necessary to adjust the `database/migrations/[date_time]_create_logs_table.php` migration to use the `log` table and not `logs`. It is also necessary to create two fields: `user_id`, an unsigned small integer, and `created_at`, a `datetime` field that will mimic the Laravel timestamp format. The code is as follows:

```
<?php

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

class CreateLogsTable extends Migration {

  /**
   * Run the migrations.
   *
   * @return void
   */
  public function up()
  {
    Schema::create('log', function(Blueprint $table)
    {
      $table->smallInteger('user_id')->unsigned();
      $table->dateTime('created_at');
    });
  }

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

Terminable middleware

In addition to performing operations after the request arrives or after the response arrives, it is possible to perform actions even after the response is sent to the browser. The class adds the `terminate` method and implements `TerminableMiddleware`:

```
use Illuminate\Contracts\Routing\TerminableMiddleware;

class StartSession implements TerminableMiddleware {

    public function handle($request, $next)
    {
        return $next($request);
    }

    public function terminate($request, $response)
    {
        // Store the session data...
    }
}
```

Logging as terminable

We could easily perform the logging of the user inside the `terminate` function as follows, since the logging could be the last action to occur in the lifecycle. The code is as follows:

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

use Closure;
use Illuminate\Contracts\Routing\TerminableMiddleware;
use MyCompany\Log;

class LogMiddleware implements TerminableMiddleware {
  /**
   * Handle an incoming request.
   *
   * @param  \Illuminate\Http\Request  $request
   * @param  \Closure  $next
   * @return mixed
   */
  public function handle($request, Closure $next)
  {
    return  $next($request);

  }
```

```
  /**
   * Terminate the request.
   *
   * @param  \Illuminate\Http\Request  $request
   * @param  \Illuminate\Http\Response $response
   */
  public function terminate($request, $response)
  {
    Log::create(['user_id'=>\Auth::user()-
>id,'created_at'=>date("Y-m-d H:i:s")]);

  }
}
```

The code has been placed into the `terminate` method, so it is outside the request-response pathway, allowing the code to remain clean.

Using middleware

If we want it that a user must be authenticated before being able to perform a certain operation, we can pass an array as the second parameter with `middleware` as the key to force the route to call the `auth` middleware on the `search` method of `AccommodationsController`:

```
Route::get('search-accommodation',
  ['middleware' => 'auth','AccommodationsController@search']);
```

In this case, the user will be redirected to the login page if not authenticated.

Route groups

Routes may be grouped together to share the same middleware. For example, if we want to protect all of the routes in our application, we can create a route group and just pass in the key-value pair `middleware` and `auth`. The code is as follows:

```
Route::group(['middleware' => 'auth'], function()
{
  Route::resource('accommodations', 'AccommodationsController');
  Route::resource('accommodations.amenities',
'AccommodationsAmenitiesController');
  Route::resource('accommodations.rooms',
'AccommodationsRoomsController');
  Route::resource('accommodations.locations',
'AccommodationsLocationsController');
  Route::resource('amenities', 'AmenitiesController');
  Route::resource('rooms', 'RoomsController');
  Route::resource('locations', 'LocationsController');
})
```

This protects every method of every route that lies inside the route group.

Multiple middleware in route groups

If even more protection is desired against nonauthenticated users, we could create a whitelist to only allow users within a certain range of IP addresses to access the application.

The following command will create the middleware that is needed:

```
$ php artisan make:middleware WhitelistMiddleware
```

The `WhitelistMiddleware` class looks like this:

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

use Closure;

class WhitelistMiddleware {
    private $whitelist = ['192.2.3.211'];
  /**
   * Handle an incoming request.
   *
   * @param  \Illuminate\Http\Request  $request
   * @param  \Closure  $next
   * @return mixed
   */
  public function handle($request, Closure $next)
  {
    if (in_array($request->getClientIp(),$this->whitelist)) {
      return $next($request);
    } else {
      return response('Unauthorized.', 401);
    }

  }
}
```

Here, a private `$whitelist` array was created with a list of the IP addresses that are set up within a company. Then, the remote port of the request is compared to the values in the array, and it is allowed to continue by returning the `$next` closure. Otherwise, an unauthorized response is returned.

Now, the `whitelist` middleware needs to be combined with the `auth` middleware. To use the `whitelist` middleware within the route group, an alias for the middleware needs to be created and inserted into the `app/Http/Kernel.php` file in the `$routeMiddleware` array. The code is as

follows:

```
protected $routeMiddleware = [
  'auth' => 'MyCompany\Http\Middleware\Authenticate',
  'auth.basic' =>
'Illuminate\Auth\Middleware\AuthenticateWithBasicAuth',
  'guest' => 'MyCompany\Http\Middleware\RedirectIfAuthenticated',
  'log' => 'MyCompany\Http\Middleware\LogMiddleware',
  'whitelist' => 'MyCompany\Http\Middleware\WhitelistMiddleware'
];
```

Next, to add this to the list of middlewares for this route group, it is necessary to substitute the string `auth` with an array, having both `auth` and `whitelist` as its contents. The code is as follows:

```
Route::group(['middleware' => ['auth','whitelist']], function()
{
  Route::resource('accommodations', 'AccommodationsController');
  Route::resource('accommodations.amenities',
            'AccommodationsAmenitiesController');
  Route::resource('accommodations.rooms',
'AccommodationsRoomsController');
  Route::resource('accommodations.locations',
'AccommodationsLocationsController');
  Route::resource('amenities', 'AmenitiesController');
  Route::resource('rooms', 'RoomsController');
  Route::resource('locations', 'LocationsController');
});
```

Now, even if the user is logged in, it will not be possible to access the protected content unless the IP address is in the whitelist.

Also, if only some of the routes are desired to be whitelisted, routes group may be nested as follows:

```
Route::group(['middleware' => 'auth', function()
{
  Route::resource('accommodations', 'AccommodationsController');
  Route::resource('accommodations.amenities',
            'AccommodationsAmenitiesController');
  Route::resource('accommodations.rooms',
'AccommodationsRoomsController');
  Route::resource('accommodations.locations',
'AccommodationsLocationsController');
  Route::resource('amenities', 'AmenitiesController');
  Route::group(['middleware' => 'whitelist'], function()
  {
```

```
    Route::resource('rooms', 'RoomsController');
  });
  Route::resource('locations', 'LocationsController');
});
```

This will require both authentication (`auth`) and whitelisting for the `RoomsController`, while all of the other controllers inside the route group will require only authentication.

Middleware exclusion and inclusion

If it is desired to perform authentication or whitelisting only for certain routes, then the constructor method should be added to the controller and the `middleware` method of the class could be used as follows:

```
<?php namespace MyCompany\Http\Controllers;

use MyCompany\Http\Requests;
use MyCompany\Http\Controllers\Controller;
use Illuminate\Http\Request;
use MyCompany\Accommodation\Room;

class RoomsController extends Controller {

  public function __construct()
  {
    $this->middleware('auth',['except' => ['index','show']);
  }
```

The first parameter is the key of the `$routeMiddleware` array in the `Kernel.php` file. The second parameter is a key and value array. The options are either `except` or `only`. The `except` option is obviously the exclusion, while the `only` option is the inclusion. In the preceding example, the `auth` middleware will be applied to all methods except for the `index` or `show` methods, which are the two read methods (they do not modify the data). Instead, if the `log` middleware should be applied on `index` and `show`, then the following constructor would be used:

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

As expected, both approaches are applied as follows and the `whitelist` middleware is added in as well:

```
public function __construct()
{
  $this->middleware('whitelist',['except' => ['index','show']);
  $this->middleware('auth',['except' => ['index','show']);
  $this->middleware('log',['only' => ['index','show']);
}
```

This code would require authentication and a whitelisted IP address for all non-

read operations, while it logs any requests to `index` and `show`.

Conclusion

Middleware can skillfully filter requests and secure the application or RESTful API from unwanted requests. It can also perform logging and redirect any requests that fall within a certain criteria.

Middleware can also provide added functionality to the existing application. For example, Laravel provides the `EncryptCookies` and `AddQueuedCookiesToResponse` middleware to deal with cookies, while `StartSession` and `ShareErrorsFromSession` deal with sessions.

The code inside `AddQueuedCookiesToResponse` does not filter the request, but rather adds to it:

```
public function handle($request, Closure $next)
  {
    $response = $next($request);
    foreach ($this->cookies->getQueuedCookies() as $cookie)
    {
      $response->headers->setCookie($cookie);
    }
    return $response;
  }
```

Summary

In this chapter, we looked at middleware, which is a useful mechanism for any functionality that should either be executed for each and every request or attached to certain routes. It is a flexible mechanism and allows the programmer to *code to an interface*, since any middleware class that implements the `Middleware` interface must include the `handle` method. Following good development principles is not only encouraged, but rather required through this type of structure.

In the next chapter, we will discuss the Eloquent ORM.

Chapter 8. Querying the Database with the Eloquent ORM

In the previous chapters, you learned how build the basic components of an application. In this chapter the Eloquent ORM, another one of the best features that makes Laravel so popular, will be introduced.

In this chapter, we'll cover the following topics:

- Basic query statements
- One-to-one, one-to-many, and many-to-many relations
- Polymorphic relations
- Eager loading

An ORM, or object relational mapping, explained in the simplest sense, turns a table into a class, its columns into attributes, and its rows into instances of that class. It creates an abstraction layer between the developer and the database and allows for easier programming, since it uses the familiar object-oriented paradigm.

We shall assume that we have a posts table with the following structure:

id	contents	author_id

To illustrate this example, the following would be the representation of a posts table:

```
<?php
namespace MyBlog;

class Post {
}
```

To add in the `id`, `contents`, and `author_id` attributes, we will add the following code to the class:

```
class Post {
    private $id;
    private $contents;
    private $author_id;
```

```
    public function getId()
    {
        return $this->id;
    }

    public function setId($id)
    {
        $this->id = $id;
    }

    public function getContents()
    {
        return $this->contents;
    }

    public function setContents($contents)
    {
        $this->contents = $contents;
    }

    public function getAuthorId()
    {
        return $this->author_id;
    }

    public function setAuthorId($author_id)
    {
        $this->author_id = $author_id;
    }

}
```

This gives us an overview of how a table may be represented by a class: the `Post` class represents an entity that has a collection of **posts**.

If the active record pattern was followed, then Eloquent can automatically manage all of the class names, key names, and their related relations. The power of Eloquent lies in its ability to give the programmer the ability to use object-oriented methods to manage the relations between the classes.

Basic operations

We will now discuss some of the basic operations. There are virtually hundreds of ways to use Eloquent, and certainly every developer will use Eloquent in the best way for their project. The following techniques are the basic building blocks upon which more complex queries may be developed.

Finding one

One of the most basic operations is to perform the following query:

```
select from rooms where id=1;
```

This is accomplished by using the `find()` method.

The `Room` facade is called with the `find` method, which accepts the ID as an argument:

```
MyCompany\Accommodation\Room::find($id);
```

Since Eloquent is based on the fluent query builder, any fluent method may be mixed and matched. Some of the fluent methods are chainable and others execute the query.

The `find()` method actually executes the query, so it always needs to be at the end of the expression.

If the ID of the model is not found, then nothing is returned. To force a `ModelNotFoundException`, which can then be trapped to perform some other operation such as logging, add `OrFail` as follows:

```
MyCompany\Accommodation\Room::findOrFail($id);
```

The where method

To query an attribute (column) other than ID, use the following command:

```
select from accommodations where name='Lovely Hotel';
```

Use the `where` method followed by the `get()` method:

```
MyCompany\Accommodation::where('name','Lovely Hotel')->get();
```

The `like` comparator may be used as follows:

```
MyCompany\Accommodation::where('name','like','%Lovely%')->get();
```

Chaining functions

Multiple where methods can be chained as follows:

```
MyCompany\Accommodation::where('name','Lovely Hotel')-
>where('city','like','%Pittsburgh%')->get();
```

The preceding command produces the following query:

```
select * from accommodations where name ='Lovely Hotel' and
description like '%Pittsburgh%'
```

Notice that if the `where` comparator is = (equals), then the second parameter (the comparator) is not needed, and the second part of the comparison is passed into the function. Also, note that an `and` operation is added between the two `where` methods. To achieve an `or` operation, the following change has to be made to the code:

```
MyCompany\Accommodation::where('name','Lovely Hotel')-
>orWhere('description','like','%Pittsburgh%')->get();
```

Notice that `or` is added to the `where` creating `orWhere()`.

Finding all

To find all of the rooms, the `all()` method is used in place of `find`. Notice that this method actually executes the query:

```
MyCompany\Accommodation\Room::all();
```

To limit the number of rooms, the `take` method is used in place of `find`. Since `take` is chainable, `get` is needed to execute the query:

```
MyCompany\Accommodation\Room::take(10)->get();
```

To achieve pagination, the following query may be used:

```
MyCompany\Accommodation\Room::paginate();
```

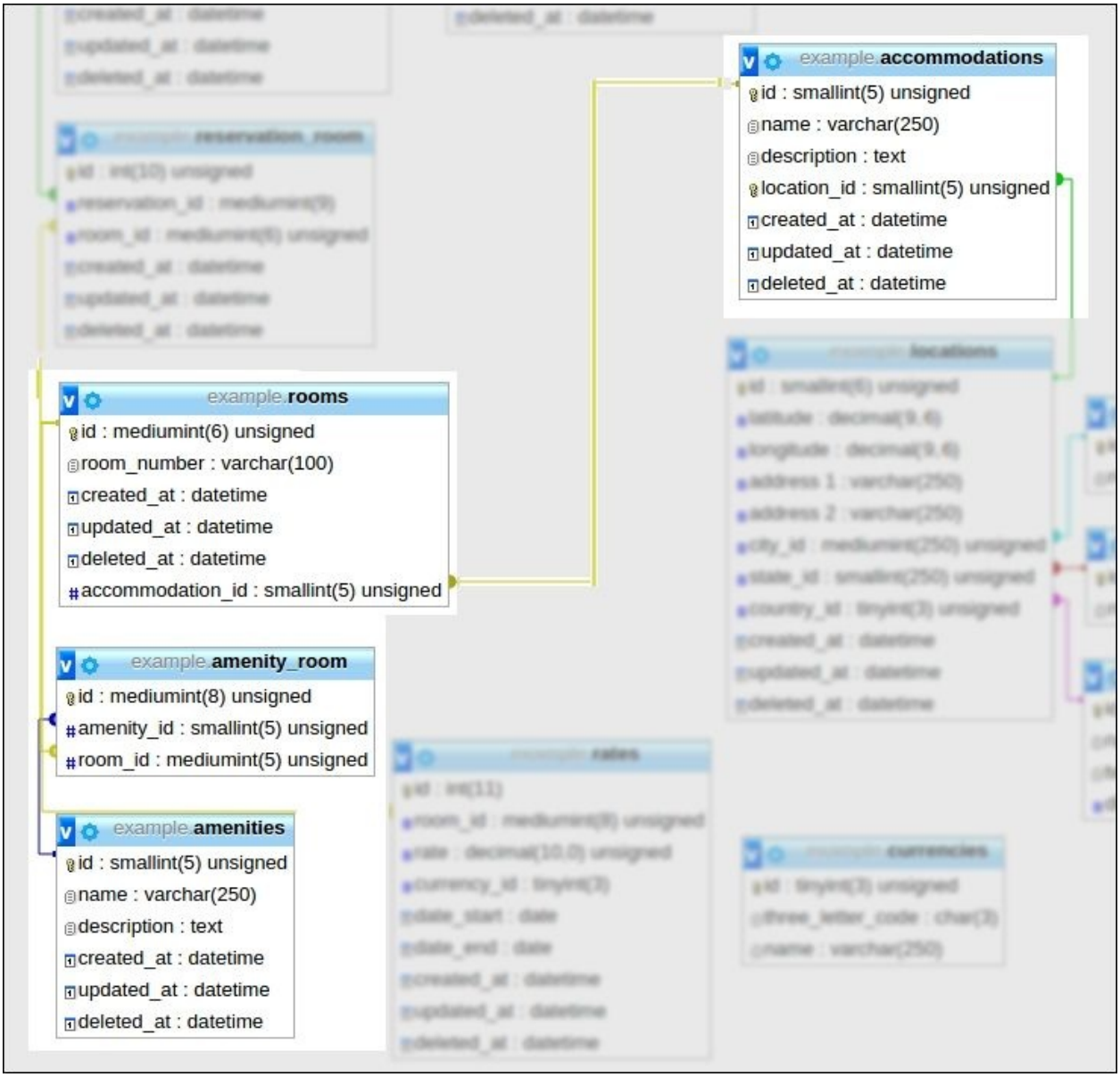
By default, the preceding query will return a JSON object as follows:

```
{"total":15,        "per_page":15,
"current_page":1,      "last_page":1,
"next_page_url":null,   "prev_page_url":null,
"from":1,        "to":15,
"data":[
{"id":9,"name":"LovelyHotel","description":"Lovely Hotel Greater
Pittsburgh","location_id":1,"created_at":null,"updated_at":
"2015-03-13 22:00:23","deleted_at":null,"franchise_id":1},
{"id":12, "name":"Grand Hotel","description":"Grand Hotel Greater
Cleveland","location_id":2,"created_at":"2015-02-
0820:09:35","updated_at":"2015-02-
0820:09:35","deleted_at":null,"franchise_id":1}
...
```

Attributes such as `total`, `per_page`, `current_page`, and `last_page` are used to give the developer an easy way to implement paging, while the array of data is returned inside of an array called `data`.

Eloquent relations

Relationships such as one-to-one, one-to-many (or many-to-one), and many-to-many are familiar to database programmers. Laravel's Eloquent has brought these concepts into an object-oriented environment. Additionally, Eloquent has even more powerful tools such as polymorphic relations, where entities can be related to more than one other entity. In the following examples, we will see the relationship between accommodations, rooms, and amenities.



One-to-one

The first relation is one-to-one. In our example software, we can use the example of a room in our accommodation. A room may only (at least easily) belong to one accommodation, so the room *belongs to* the accommodation. Inside the `Room` Eloquent model, the following code tells Eloquent that the room belongs to the `accommodation` function:

```
class Room extends Eloquent {
     public function accommodation()
     {
         return $this->belongsTo('MyCompany\Accommodation');
     }
}
```

Sometimes, the database tables do not follow the active record pattern, especially if the programmer inherits a legacy database. If the database used a table called `bedroom` instead of `rooms`, then the class would add an attribute to indicate the table name:

```
class Room extends Eloquent {
    protected $table = 'bedroom';
}
```

When the following route code is executed, then the `accommodation` object will be returned as a JSON object:

```
Route::get('test-relation',function(){
    $room = MyCompany\Accommodation\Room::find(1);
    return $room->accommodation;
});
```

The response will be as follows:

```
{"id":9,"name":"LovelyHotel","description":"Lovely Hotel Greater
Pittsburgh","location_id":1,"created_at":null,"updated_at":
"2015-03-13 22:00:23","deleted_at":null}
```

Tip

One common mistake is to use the following command:

```
return $room->accommodation();
```

In this case, the programmer expects to return the model. This will return the

actual `belongsTo` relation and in the context of the RESTful API, there will be an error thrown:

```
Object of class Illuminate\Database\Eloquent\Relations\BelongsTo
could not be converted to string
```

This is because Laravel can convert the JSON object to a string, but not a relationship.

The SQL run is as follows:

```
select * from rooms where rooms.id = '1' limit 1
select * from accommodations where accommodations.id = '9' limit
1
```

Eloquent tends to favor multiple simpler queries as opposed to doing larger joins.

First the room is found. Then, `limit 1` is added because `find` is only used to find a single entity or row. Once the `accommodation_id` is found, the next query will find the accommodation with that corresponding ID and return the object. If the active record pattern was followed, the SQL that Eloquent produces is extremely readable.

One-to-many

The second relation is one-to-many. In our example software, we can use the example of an accommodation having many rooms. Since rooms may belong to one accommodation, then the the accommodation has *many rooms*. Inside the `Accommodation` Eloquent model, the following code tells Eloquent that the accommodation has many rooms.

```
class Accommodation {
    public function rooms(){
        return $this->hasMany('\MyCompany\Accommodation\Room');
    }
}
```

In a similar route, the following code is run. This time, a collection of `rooms` objects will be returned as JSON-formatted objects inside an array:

```
Route::get('test-relation',function(){
    $accommodation = MyCompany\Accommodation::find(9);
    return $accommodation->rooms;
});
```

The response will be the following array:

```
[{"id":1,"room_number":0,"created_at":null,"updated_at":null,
"deleted_at":null,"accommodation_id":9},{"id":3,"room_number":
12,"created_at":"2015-03-14 08:52:25","updated_at":"2015-03-14
08:52:25","deleted_at":null,"accommodation_id":9},{"id":6,
"room_number":12,"created_at":"2015-03-14
09:03:36","updated_at":"2015-03-14
09:03:36","deleted_at":null,"accommodation_id":9},{"id":
14,"room_number":12,"created_at":"2015-03-14
09:26:36","updated_at":"2015-03-
1409:26:36","deleted_at":null,"accommodation_id":9}]
```

The SQL run is as follows:

```
select * from accommodations where accommodations.id = ? limit 1
select * from rooms where rooms.accommodation_id = '9' and
rooms.accommodation_id is not null
```

As before, the accommodation is found. The second query will find the rooms that belong to that accommodation. A check is added to confirm that the `accommodation_id` is not null.

Many-to-many

In our example software application, the relationship between amenity and room is many-to-many. Each room can have many amenities, such as Internet access and a Jacuzzi, and each amenity is shared among many rooms: *every room in an accommodation could and should have internet access!* The following code, which uses a `belongsToMany` relationship, enables an amenity to belong to many rooms:

```
class Amenity {
  public function rooms(){
        return $this-
>belongsToMany('\MyCompany\Accommodation\Room');
    }
}
```

The test route, which tells us how each room has a certain amenity, is written as follows:

```
Route::get('test-relation',function(){
    $amenity = MyCompany\Accommodation\Amenity::find(3);
    return $amenity->rooms;
});
```

A list of rooms is returned:

```
[{"id":1,"room_number":0,"created_at":2015-03-14
08:10:45,"updated_at":null,"deleted_at":null,
"accommodation_id":9},{"id":5,"room_number":12,
"created_at":"2015-03-14 09:00:38","updated_at":"2015-03-14",
09:00:38","deleted_at":null,"accommodation_id":12},
...]
```

The SQL executed is as follows:

```
select * from amenities where amenities.id = ? limit 1
select rooms.*, amenity_room.amenity_id as pivot_amenity_id,
amenity_room.room_id as pivot_room_id from rooms inner join
amenity_room on rooms.id = amenity_room.room_id where
amenity_room.amenity_id = 3
```

We recall the `belongToMany` relationship that returns the rooms that have a particular amenity:

```
class Amenity {
```

```
   public function rooms(){
        return $this-
>belongsToMany('\MyCompany\Accommodation\Room');
    }
}
```

Eloquent skillfully gives us the corresponding `belongsToMany` relationship to determine which amenities a particular room has. The syntax is exactly the same:

```
class Room {
     public function amenities(){
         return $this-
>belongsToMany('\MyCompany\Accommodation\Amenity');
     }
 }
```

The test route is virtually the same, just substituting `amenities` for `rooms`:

```
Route::get('test-relation',function(){
    $room = MyCompany\Accommodation\Room::find(1);
    return $room->amenities;
});
```

The result is a list of amenities for the room with ID 1:

```
[{"id":1,"name":"Wifi","description":"Wireless Internet
Access","created_at":"2015-03-1409:00:38","updated_at":"2015-03-
14 09:00:38","deleted_at":null},{"id":2,"name":
"Jacuzzi","description":"Hot tub","created_at":"2015-03-14
09:00:38","updated_at":null,"deleted_at":null},{"id":3,"name":
"Safe","description":"Safe deposit box for protecting
valuables","created_at":"2015-03-1409:00:38","updated_at": "2015-
03-1409:00:38","deleted_at":null}]
```

The query used is as follows:

```
select * from rooms where rooms.id = 1 limit 1
select amenities.*, amenity_room.room_id as pivot_room_id,
amenity_room.amenity_id as pivot_amenity_id from amenities inner
join amenity_room on amenities.id = amenity_room.amenity_id where
amenity_room.room_id = '1'
```

The query, substituting `room_id` for `amenity_id` and `rooms` for `amenities`, is clearly parallel.

Has-many-through

One great feature of Eloquent is "has-many-through". What if the requirements of the software change and we are asked to group some of the accommodations together into franchises? If an application user would like to search for a room, any of the rooms in any of the accommodations that belong to that franchise could be found. A franchises table will be added, and a nullable column to the accommodations table called `franchise_id` will be added. This will optionally allow for an accommodation to belong to a franchise. Rooms already belong to accommodations through the `accommodation_id` column.

A room belongs to an `accommodation` through its `accommodation_id` key, while an accommodation belongs to a franchise through its `franchise_id` key.

Eloquent allows us to retrieve the rooms associated to a franchise by using `hasManyThrough`:

```
<?php namespace MyCompany;

use Illuminate\Database\Eloquent\Model;

class Franchise extends Model {

    public function rooms()
    {
        return $this-
>hasManyThrough('\MyCompany\Accommodation\Room',
'\MyCompany\Accommodation');
    }
}
```

The `hasManyThrough` relationship takes the target or the "has" as its first parameter (in this case, the room) and its "through" as the second parameter (in this case, the accommodation).

The logic stated as a phrase is: *This franchise has many rooms through its accommodations*.

Using the previous test route, the code is written as follows:

```
Route::get('test-relation',function(){
    $franchise = MyCompany\Franchise::find(1);
    return $franchise->rooms;
```

```
});
```

The rooms are returned as an array as would be expected:

```
[{"id":1,"room_number":0,"created_at":null,"updated_at":null,"del
eted_at":null,"accommodation_id":9,"franchise_id":1},
{"id":3,"room_number":12,"created_at":"2015-03-14
08:52:25","updated_at":"2015-03-14
08:52:25","deleted_at":null,"accommodation_id":9,
"franchise_id":1},{"id":6,"room_number":12,"created_at":"2015-03-
14 09:03:36","updated_at":"2015-03-14
09:03:36","deleted_at":null,"accommodation_id":9,
"franchise_id":1},
]
```

The queries executed are as follows:

```
select * from franchises where franchises.id = ? limit 1
select rooms.*, accommodations.franchise_id from rooms inner join
accommodations on accommodations.id = rooms.accommodation_id
where accommodations.franchise_id = 1
```

Polymorphic relations

One great feature of Eloquent is the possibility to have an entity whose relationship is polymorphic. The two parts of the word, *poly* and *morphic*, are from the Greek language. Since *poly* means *many* and *morphic* means *shape*, we can now easily imagine a relationship having multiple forms.

Amenitiable relationships

An amenity in our example software is something that is associated with a room, such as a Jacuzzi. Certain amenities, such as covered parking or an airport shuttle service, could also relate to an accommodation itself. We could create two pivot tables for this, one called `amenity_room` and another called `accommodation_amenity`. Another great way to do this is to combine the two into one table and use a field to distinguish between the two types or relationship.

To do this, we will need a field to distinguish between *amenity and room* and *amenity and room*, something we could call a relationship type. Laravel's Eloquent skillfully handles this automatically.

Eloquent uses the suffix `-able` to make this happen. In our example, we would create a table that has the following fields:

- `id`
- `name`
- `description`
- `amenitiable_id`
- `amenitiable_type`

The first three fields are familiar, but two new fields added. One of them will contain the ID of either the accommodation or the room.

The Amenity table structure

For example, given a room with ID 5, `amenitiable_id` will be `5` while `amenitiable_type` will be `Room`. Given an accommodation with ID 5, `amenitiable_id` will be `5` while `amenitiable_type` will be `Accommodation`:

id	name	description	amenitiable_id	amenitiable_type
1	Wireless internet	Internet conn.	5	Room
2	Covered parking	Parking in garage	5	Accommodation
3	Sea view	Ocean view from room	5	Room

The Amenity model

In terms of code, the `Amenity` model will now contain an "amenitiable" function:

```
<?php
namespace MyCompany\Accommodation;

use Illuminate\Database\Eloquent\Model;

class Amenity extends Model
{
    public function rooms(){
        return $this-
>belongsToMany('\MyCompany\Accommodation\Room');
    }
    public function amenitiable()
    {
        return $this->morphTo();
    }
```

The Accommodation model

The `Accommodation` model will change the `amenities` method to use `morphMany` instead of `hasMany`:

```
<?php namespace MyCompany;

use Illuminate\Database\Eloquent\Model;

class Accommodation extends Model {
    public function rooms(){
        return $this->hasMany('\MyCompany\Accommodation\Room');
    }

    public function amenities()
    {
        return $this-
>morphMany('\MyCompany\Accommodation\Amenity', 'amenitiable');
    }
}
```

The Room model

The `Room` model will contain the same `morphMany` method:

```
<?php
```

```
namespace MyCompany\Accommodation;

use Illuminate\Database\Eloquent\Model;

class Room extends Model
{
    protected $casts = ['room_number'=>'integer'];
    public function accommodation(){
        return $this->belongsTo('\MyCompany\Accommodation');
    }
    public function amenities() {
        return $this-
>morphMany('\MyCompany\Accommodation\Amenity', 'amenitiable');
    }

}
```

Now, when the amenities are requested for a room or an accommodation, Eloquent will automatically distinguish between them:

```
$accommodation->amenities();
$room->amenities();
```

Each of these functions returns the correct type of amenity for room and for accommodation.

Many-to-many polymorphic relations

It is possible, though, that some amenities could be shared between a room and an accommodation. In this case, a many-to-many polymorphic relation is used. The pivot table now adds several fields:

amenity_id	amenitiable_id	amenitiable_type
1	5	Room
1	5	Accommodation
2	5	Room
2	5	Accommodation

As illustrated, both the room with ID 5 and the accommodation with ID 5 have amenities with IDs 1 and 2.

Has relationships

If we would like to select all of the accommodations that are associated to a franchise, the `has()` method is used, where the relation is passed as the parameter:

```
MyCompany\Accommodation::has('franchise')->get();
```

We will get the following JSON array:

```
[{"id":9,"name":"LovelyHotel","description":"Lovely Hotel Greater
Pittsburgh","location_id":1,"created_at":null,"updated_at":
"2015-03-13 22:00:23","deleted_at":null,"franchise_id":1},
{"id":12,"name": "Grand Hotel","description":"Grand Hotel Greater
Cleveland","location_id":2,"created_at": "2015-02-
0820:09:35","updated_at": "2015-02-
0820:09:35","deleted_at":null,"franchise_id":1}]
```

Notice that the `franchise_id` value is 1, which means the accommodations have a franchise associated with them. Optionally, a `where` may be added to the `has` creating a `whereHas` function. The code is as follows:

```
MyCompany\Accommodation::whereHas('franchise',
                  function($query){
      $query->where('description','like','%Pittsburgh%');
      })->get();
```

Notice that `whereHas` takes a closure as its second parameter.

This would return only the accommodations where the description contains `Pittsburgh`, so the returned array would contain only results like this:

```
[{"id":9,"name":"LovelyHotel","description":"Lovely Hotel Greater
Pittsburgh","location_id":1,"created_at":null,"updated_at":
"2015-03-13 22:00:23","deleted_at":null,"franchise_id":1}]
```

Eager loading

Another great mechanism that Eloquent provides is eager loading. If we want return all of the franchises together with all of their accommodations, we simply need to add an `accommodations` function to our `Franchise` model as follows:

```
    public function accommodations()
    {
        return $this->hasMany('\MyCompany\Accommodation');
    }
```

Then, by adding a `with` clause to the statement, the accommodations are returned for each franchise:

```
MyCompany\Franchise::with('accommodations')->get();
```

We can also list the rooms associated with each accommodation as follows:

```
MyCompany\Franchise::with('accommodations','rooms')->get();
```

If we want to return the rooms nested inside of the accommodation array, then the following syntax should be used:

```
MyCompany\Franchise::with('accommodations','accommodations.rooms'
) ->get();
```

We will get the following output:

```
[{"id":1,"accommodations":
[
{"id":9,
"name":"Lovely Hotel",
"description":"Lovely Hotel Greater Pittsburgh",
"location_id":1,
"created_at":null,
"updated_at":"2015-03-13 22:00:23",
"deleted_at":null,
"franchise_id":1,
"rooms":[{"id":1,"room_number":0,"created_at":null,"updated_at":
null,"deleted_at":null,"accommodation_id":9},
]},
{"id":12,"name":"GrandHotel","description":"Grand Hotel Greater
Cleveland","location_id":2,"created_at":"2015-02-08…
```

In this example, `rooms` is contained within `accommodation`.

Conclusion

Laravel's ORM is powerful. In fact, there are too many types of operations to list within a single book. The simplest queries can be accomplished with a few keystrokes.

Laravel's Eloquent command gets converted into fluent commands, so if something more complicated is desired, then the fluent syntax may be used. If a very complicated query needs to be performed, then the `DB::raw()` function can even be used. This will allow exact strings to be used inside the query builder. Here is an example:

```
$users = DB::table('accommodation')
                      ->select(DB::raw('count(*) as
number_of_hotels'))->get();
```

This returns just the number of hotels:

```
[{"number_of_hotels":15}]
```

Learning to design the software, starting with the domain and then thinking about which entities are involved in that domain, will help a developer think in an object-oriented manner. Having a list of entities leads to the creation of the table, so the actual creation of the schema will be performed last. This approach may take some getting used to. Understanding Eloquent relationships is key to being able to produce expressive, readable statements that query the database while hiding away the complexity.

Another reason why Eloquent is extremely useful is in the case of a legacy database. If the ORM is applied in a situation where the tables have nonstandard names, the keys are not named by the same name, or the column names are simply not easily understandable, Eloquent provides the developer with tools to actually help homogenize the table names and field names, and perform the relations by providing attribute getters and setters.

For example, if the field names are `fname1` and `fname2`, we can use a get attribute function inside our model, where the syntax is `get` followed by the desired name to use in the application and the attribute. So, in the case of `fname1`, the function would be added as follows:

```
public function getUsernameAttribute($value)
```

```
{
  return $this->attributes['fname1'];
}
```

Functions such as these are Eloquent's real selling point. In this chapter, you learned how to find data inside your database by using entity models, limiting the results through the addition of `where`, relationships, powerful conventions such as polymorphic relationships, and auxiliary helpers such as pagination.

Summary

In this chapter, the Eloquent ORM was demonstrated in detail. Eloquent is an object-oriented wrapper to what actually happens between the database and the code. Since the Fluent query builder is easily accessible, it is important to be familiar with how the queries are written. This will help in debugging and also cover complex cases where Eloquent is not adequate. In this chapter, most of the concepts of Eloquent have been discussed. However, there are many more methods available, so further reading is encouraged.

In the next chapter, among other topics, you will learn how to scale the database to perform better on a larger scale.

Chapter 9. Scaling Laravel

A characteristic of frameworks built in any programming language is the use of various components. As we saw in the previous chapters, frameworks provide a software developer with many different prebuilt tools to accomplish tasks such as authentication, database interaction, and RESTful API creation.

However, regarding frameworks, scalability issues are always the worst fears of any manager in the information technology field. As with any library that uses existing code, there will be always some amount of overhead, some amount of bloat, something more than is actually needed.

Scalability issues

There are many reasons why a framework could not easily scale. Let's look at a short list of issues:

- One issue is unnecessary code and packages that are not directly related to the actual application being built. Not every project needs authentication, for example, a database drivers per MySQL. Packages that are part of the core of the framework must be monitored for compatibility issues.
- Design patterns, opinions, and learning curves often prevent new team members from getting familiar quickly. As the project expands, the day-to-day development needs to grow, and a software development group must continually recruit members who are either already somewhat familiar with a framework or at least understand its basic concepts.
- Framework security issues require continual monitoring of the framework community's website or repository to gather information regarding emergency security updates that are required. Even the underlying web server and operating system itself need to be monitored. At the time of writing, Laravel 5.1 is nearing its release and it will require PHP 5.5, since PHP 5.4 will be declared end-of-life later in 2015.
- ORMs such as Eloquent always add a slight amount of overhead, since the code needs to be converted first from Eloquent into the fluent query builder and then into PDO code. Obviously, using an object-oriented approach to query the database is a wise choice, but it has a cost.

Towards the enterprise

Despite possible roadblocks, Laravel will continue to be a strong option in the future of enterprise. PHP 7 is going to be very fast, and frameworks such as Zend Framework 3 have already announced PHP 7 optimization in their road map. Also, by using the **FastCGI Process Manager** (**FPM**), NGINX web server, and allowing PHP's caching mechanisms to work properly, application scalability will continue to become more accepted in the enterprise space as its renaissance continues and new developers contribute to its core.

In this chapter, you will learn how to allow Laravel to perform better in the enterprise setting where scalability issues are paramount. First, router caching will be discussed. Then, you will learn about the many tools, techniques, and even new microframeworks that are being developed with scalability at the forefront. Specifically, we will discuss an official microframework derived from Laravel, **Lumen**. Finally, you will learn how to use the database efficiently through a technique called *read* and *write*.

In terms of the size of the code base, Laravel has one of the smallest code bases as compared to Zend or Symfony, although it does use some Symfony components. As mentioned in previous chapters, different packages were removed to lighten up the footprint, taking cues from Symfony's component-based ideology. For example, the HTML, SSH, and annotations packages are no longer included by default.

Route caching

Route caching helps to speed things up. In Laravel 5, a caching mechanism was introduced to speed up the execution.

The example `routes.php` is shown here:

```
Route::post('reserve-room', 'ReservationController@store');

Route::controllers([
  'auth' => 'Auth\AuthController',
  'password' => 'Auth\PasswordController',
]);
Route::post('/bookRoom','ReservationsController@reserve',
['middleware' => 'auth', 'domain'=>'booking.hotelwebsite.com']);

Route::resource('rooms', 'RoomsController');

Route::group(['middleware' => ['auth','whitelist']], function()
{

  Route::resource('accommodations', 'AccommodationsController');
  Route::resource('accommodations.amenities',
'AccommodationsAmenitiesController');
  Route::resource('accommodations.rooms',
'AccommodationsRoomsController');
  Route::resource('accommodations.locations',
'AccommodationsLocationsController');
  Route::resource('amenities', 'AmenitiesController');
  Route::resource('locations', 'LocationsController');
});
```

By running the following command, Laravel will cache the routes:

```
$ php artisan route:cache
```

Then, place them into the following directory:

```
/vendor/routes.php
```

Here is a small portion of the resultant file:

```
<?php

/*
```

```
| Load The Cached Routes
|--------------------------------------------------------------------
----------
|
| Here we will decode and unserialize the RouteCollection
instance that
| holds all of the route information for an application. This
allows
| us to instantaneously load the entire route map into the
router.
|
*/

app('router')->setRoutes(
  unserialize(base64_decode('TzozNDoiSWxsdW1pbmF0ZVxSb3V0aW5nXF
JvdXRlQ29sbGVjdGlvbiI6NDp7czo5OiIAKgByb3V0ZXMiO2E6Njp7czozOiJH
RVQiO2E6NTA6e3M6MToiLyI7TzoyNDoiSWxsdW1pbmF0ZVxSb3V0aW5nXFJvdX
RlIjo3OntzOjY6IgAqAHVyaSI7czoxOiIvIjtzOjEwOiIAKgBtZXRob2RzIjth
OjI6e2k6MDtzOjM6IkdFVCI7aToxO3M6NDoiSEVBRCI7fX
...
Db250cm9sbGVyc1xBbWVuaXRpZXNDb250cm9sbGVyQHVwZGF0ZSI7cjoxNDQx
O3M6NTQ6Ik15Q29tcGFueVxIyb2xsZXJzXEhvdGVsQ29udHJvbGxlckBkZXN0c
m95IjtyOjE2MzI7fX0='))
);
```

As the DocBlock states, the routes are encoded in base64 and then serialized:

```
unserialize(base64_decode( … ));
```

This performs some precompilation. If we base64 decode the contents of the file, we obtain the serialized data. The following code is an extract of the file:

```
O:34:"Illuminate\Routing\RouteCollection":4:{s:9:"*routes"; a:6:
{s:3:"GET";a:50:{s:1:"/";O:24:"Illuminate\Routing\Route": 7:
{s:6:"*uri";s:1:"/";s:10:"*methods";a:2:{i:0;s:3:"GET";i:1;
s:4:"HEAD";}s:9:"*action";a:5:{s:4:"uses";s:50:"MyCompany
\Http\Controllers\WelcomeController@index";s:10:"controller";
s:50:"MyCompany\Http\Controllers\WelcomeController@index";
s:9:"namespace";s:26:"MyCompany\Http\Controllers";s:6:"prefix";
N;s:5:"where";a:0:{}}s:11:"*defaults";a:0:{}s:9:"*wheres"; a:0:
{}s:13:"*parameters";N;s:17:"*parameterNames";N;
}s:4:"home";O:24:"Illumin…

"MyCompany\Http\Controllers\HotelController@destroy";r:1632;}}
```

If the `/vendor/routes.php` file exists, it is used instead of the `routes.php` file that is located at `/app/Http/routes.php`. If at some point using the route caching file is no longer desired, use the following `artisan` command:

```
$ php artisan route:clear
```

This command will delete the cached `routes` file and Laravel will begin using the `/app/Http/routes.php` file again.

Tip

It is important to note that if there are any closures used in the `routes.php` file, then caching will fail. Here is an example of a closure in a route:

```
Route::get('room/{$id}', function(){
  return Room::find($id);
});
```

It is inadvisable to use closure in the `routes.php` file for any reason. To be able to use route caching, relocate the code used within the closure into a controller.

Illuminate routing

All of this work speeds up an important part of the request life cycle, the routing. In Laravel, the routing class is located inside the `illuminate/routing` namespace:

```
<?php namespace Illuminate\Routing;
use Closure;
use LogicException;
use ReflectionFunction;
use Illuminate\Http\Request;
use Illuminate\Container\Container;
use Illuminate\Routing\Matching\UriValidator;
use Illuminate\Routing\Matching\HostValidator;
use Illuminate\Routing\Matching\MethodValidator;
use Illuminate\Routing\Matching\SchemeValidator;
use Symfony\Component\Routing\Route as SymfonyRoute;
use Illuminate\Http\Exception\HttpResponseException;
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;
```

Examining the `use` operators, it is clear that the routing mechanism consists of quite a few classes. The most important line is as follows:

```
use Symfony\Component\Routing\Route as SymfonyRoute;
```

Laravel uses Symfony's routing class. However, a new routing package written by Nikita Popov has emerged. `FastRoute` is a fast request router that is faster than other routing packages and addresses some of the issues of the existing routing packages. This component is one of the major advantages of the of the Lumen microframework.

Lumen

In soda marketing terms, Lumen could be considered Laravel *Light* or Laravel *Zero*. In addition to using the `FastRoute` routing package, many packages have been removed from Lumen, allowing it to be minimal and reduce its footprint.

Comparison between Laravel and Lumen

The packages in Laravel and Lumen were compared and listed in the following table. These packages are installed when the following command is run:

```
$ composer update –-no-dev
```

The preceding command is used when the development is completed and the application is ready for deployment on the server. Tools such as PHPUnit and PHPSpec are obviously excluded at this stage.

The package names are aligned to illustrate where the packages are present in both Laravel and Lumen:

Laravel Packages	**Lumen Packages**
-	nikic/fast-route
illuminate/cache	-
illuminate/config	illuminate/config
illuminate/console	illuminate/console
illuminate/container	illuminate/container
illuminate/contracts	illuminate/contracts
illuminate/cookie	illuminate/cookie
illuminate/database	illuminate/database
illuminate/encryption	illuminate/encryption
illuminate/events	illuminate/events

illuminate/exception	_
illuminate/filesystem	illuminate/filesystem
illuminate/foundation	_
illuminate/hashing	illuminate/hashing
illuminate/http	illuminate/http
illuminate/log	_
illuminate/mail	_
illuminate/pagination	illuminate/pagination
illuminate/pipeline	_
illuminate/queue	illuminate/queue
illuminate/redis	_
illuminate/routing	_
illuminate/session	illuminate/session
illuminate/support	illuminate/support
illuminate/translation	illuminate/translation
illuminate/validation	illuminate/validation

illuminate/view	illuminate/view
jeremeamia/superclosure	-
league/flysystem	-
monolog/monolog	monolog/monolog
mtdowling/cron-expression	mtdowling/cron-expression
nesbot/carbon	-
psy/psysh	-
swiftmailer/swiftmailer	-
symfony/console	-
symfony/css-selector	-
symfony/debug	-
symfony/dom-crawler	-
symfony/finder	-
symfony/http-foundation	symfony/http-foundation
symfony/http-kernel	symfony/http-kernel

symfony/process	-
symfony/routing	-
symfony/security-core	symfony/security-core
symfony/var-dumper	symfony/var-dumper
vlucas/phpdotenv	-
classpreloader/classpreloader	-
danielstjules/stringy	-
doctrine/inflector	-
ext-mbstring	-
ext-mcrypt	-

At the time of writing, there are 51 packages (shown in the left column) that are installed in Laravel 5.0 using the nondevelopment configuration. Compare this number of packages to the number of packages installed in Lumen (shown in the right column)—there are just 24.

The aforementioned `nikic/fast-route` package is the only package that Lumen has which Laravel does not. The `symfony/routing` package is the complimentary package found in Laravel.

Lean application development

We will use an example, a simple public-facing RESTful API. This RESTful API displays the names and addresses of a list of accommodations in JSON format to any user via `GET`:

- If no passwords are to be used, then `ext/mcrypt` is not needed.
- If no date calculations are to performed, then `nesbot/carbon` is not needed. Since there is no HTML interface, then the following libraries involved in testing the HTML of an application, `symfony/css-selector` and `symfony/dom-crawler`, will not be needed.
- If no e-mail is to be sent to the user, then neither `illuminate/mail` nor `swiftmailer/swiftmailer` is needed.
- If no special interaction with the filesystem is needed, then there is no need for `league/flysystem`.
- If not commands that are to run from the command line, then the `symfony/console` is not needed.
- If Redis is not needed, then `illuminate/redis` may be left out.
- If specific configuration values will not be needed for different environments, then `vlucas/phpdotenv` is not needed.

Tip

The `vlucas/phpdotenv` package is a suggested package in the `composer.json` file.

It is clear that the decision to remove certain packages has been done carefully so as to lighten up Lumen as needed with the simplest of applications in mind.

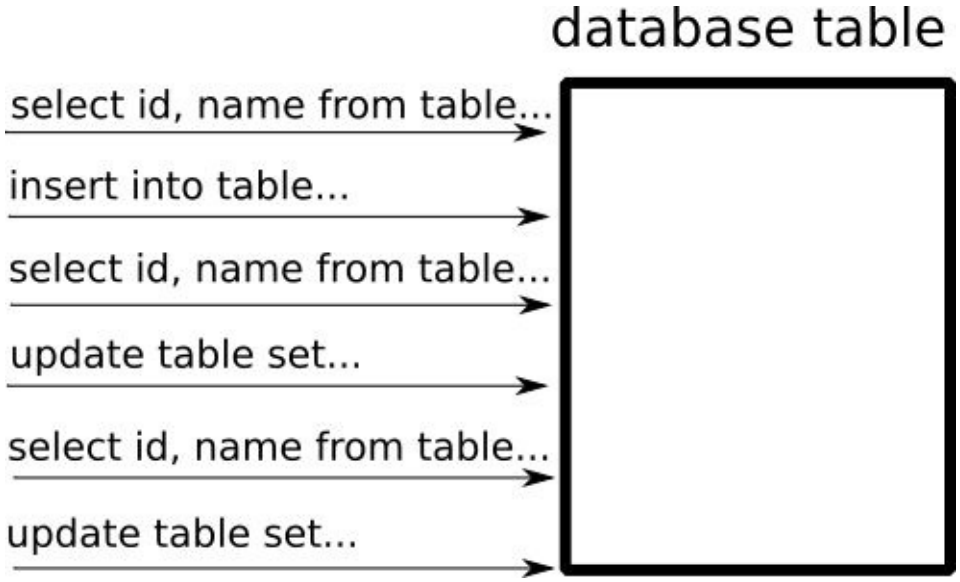
Read/write

Laravel has another great mechanism for helping its performance in the enterprise: read and write. This is related to database performance, but the functionality is so easy to set up that any application can take advantage of its usefulness.

Regarding MySQL, the original MyISAM database engine required a lock on the entire table during inserts, updates, and deletes. This caused massive bottlenecks during large operations that modified the data and select queries waited to access these tables. With the introduction of InnoDB, `UPDATE`, `INSERT`, and `DELETE` SQL statements required a lock only at the row-level. This tremendously impacted performance, since selects could read from various parts of a table where other operations were taking place.

MariaDB, a MySQL fork, claims faster performance than the traditional MySQL. Replacing the database engine with TokuDB will give even higher performance, especially in a big data setting.

Another mechanism to speed up performance of the database is the use of a master/slave configuration. In the following diagram, all of the operations are performed on a single table. The inserts and updates will lock down single rows and the select statements will be performed as allocated.



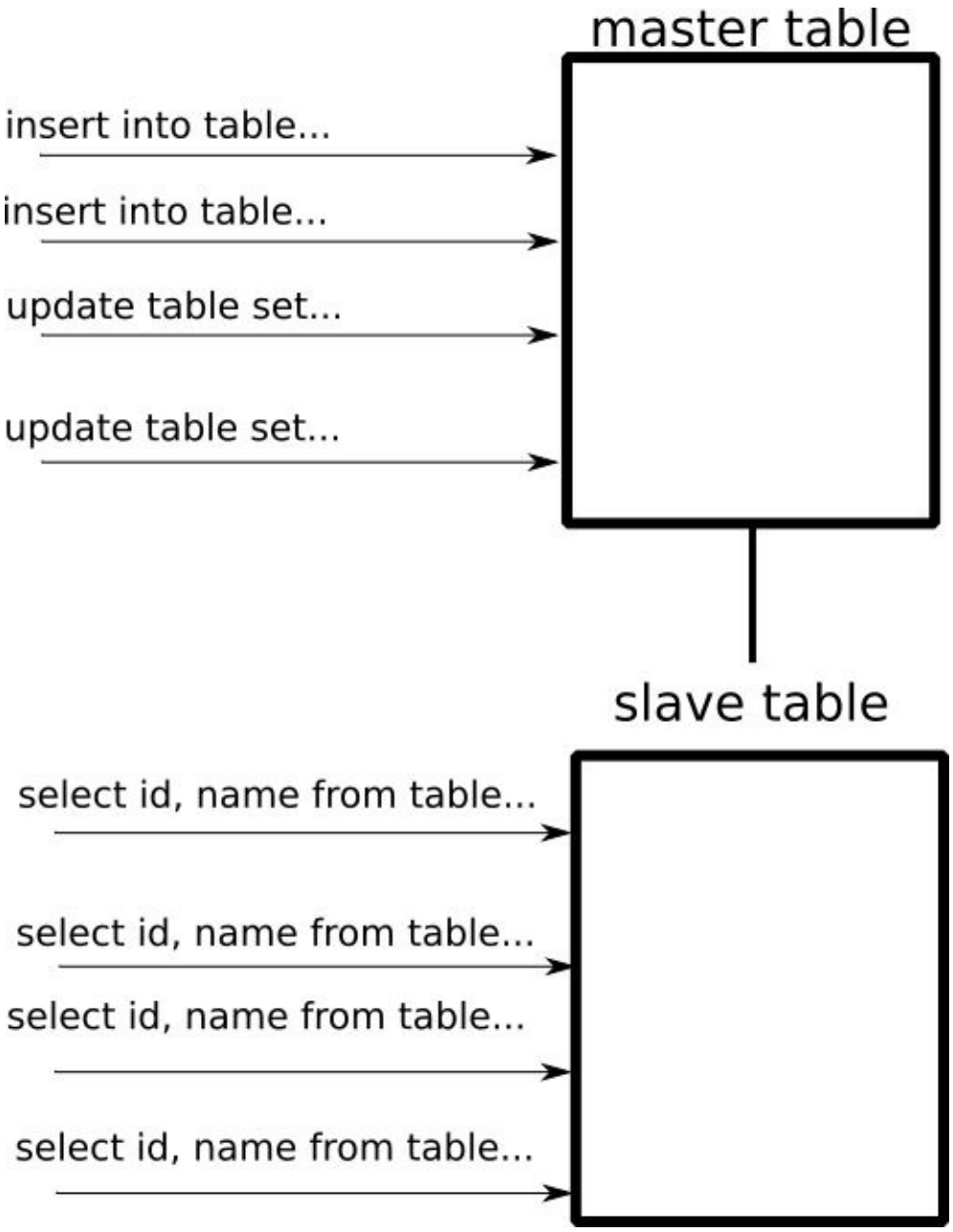
Traditional database table actions

Master table

The master/slave configuration uses a master table that allows `SELECT`, `UPDATE`, and `DELETE` statements. These statements modify the table or write to it. There may also be multiple master tables. Each master table is kept continually synchronized: changes made to any table needs to be communicated to the master table.

Slave table

The slave table is a slave to the master. It depends on the master table for its changes. SQL clients can only perform read operations (`SELECT`) from it. There may also be multiple slaves that depend on one or more multiple master tables. The master table communicates all of its changes to all the slaves. The following diagram shows the basic architecture of a master/slave setup:



Master and slave (read/write setup)

This continual synchronization adds slight overhead to the database structure; however, it presents important advantages:

Since only `SELECT` statements can be performed on the slave table while `INSERT`, `UPDATE`, and `DELETE` statements can be performed on the master table, the slave table is free to accept many `SELECT` statements freely, without having to "wait" for any operations involving the same rows to finish.

An example of this would be a currency exchange rate or stock price table. This table would be continually updated in real time with the latest values, possibly even many times per second. Obviously, a website that allows many users to access this information could potentially have thousands of visitors. Also, the web page used to display this data may make continual multiple requests per user.

Performing many `SELECT` statements would be slightly slower when there are `UPDATE` statements that need to access the same data at the same time.

By using a master/slave configuration, the `SELECT` statements would be performed only on the slave table. This table receives only the data that has changed in an extremely optimized way.

In plain PHP using a library such as `mysqli`, there could be two database connections configured:

```
$master=mysqli_connect('127.0.0.1:3306','dbuser','dbpassword','my
database');
$slave=mysqli_connect('127.0.0.1:3307','dbuser','dbpassword','myd
atabase');
```

In this simplified example, the slave is set up on the same server machine. In a real application, it would most likely be set up on another server machine to take advantage of separate hardware.

Then, all of the SQL statements which involve a *write* statement would be performed on the slave and *read* would be performed on the master.

This would add some overhead to the programming efforts, as a different

connection would need to be passed into each SQL statement:

```
$result= mysqli_real_query($master,"UPDATE exchanges set
rate='1.345' where exchange_id=2");
$result= mysqli_query($slave,"SELECT rate from exchanges where
exchange_id=2");
```

In the preceding code example, it would be prudent to remember which SQL statements should be used for the master and which SQL statements should be used for the slave.

Configuring read/write

As stated before, code written in Eloquent is converted into fluent query-builder code. This code is then converted to PDO, which is a standard wrapper around the various database drivers.

Laravel provides the ability to manage master/slave configurations though its read/write configuration. This allows programmers to write Eloquent and fluent query-builder code without having to worry about whether the queries will be executed on the master or slave table. Also, a software project that starts out with a non-master/slave configuration and later needs to scale up to a master/slave setup will only need to change one aspect of the database configuration. The database configuration file is located at `config/database.php`.

As an element of the `connections` array, an entry with the key `mysql` will be created with the following configuration:

```
'connections' =>
'mysql' => [
    'read' => [
        'host' => '192.168.1.1',

     'password'  => 'slave-Passw0rd',
    ],
    'write' => [
        'host' => '196.168.1.2',

    'username'  => 'dbhostusername'
    ],
    'driver'    => 'mysql',
    'database'  => 'database',
    'username'  => 'dbusername',
    'password'  => 's0methingSecure',
    'charset'   => 'utf8',
    'collation' => 'utf8_unicode_ci',
    'prefix'    => '',
],
```

The read and write represent slave and master respectively. Since the parameters cascade, if the username, password, and database name are the same, then only the IP address of the host name needs to be listed. However, any values can be overridden. In this example, the read has a password that is different from that of the master and the write has a username that is different from the slave.

Creating a master/slave database configuration

To set up a master/slave database, perform the following steps from the command line.

1. The first step is to determine which address the MySQL server is bound to. To do this, locate the line of the MySQL configuration file that contains the bind-address parameter:

   ```
   bind-address            = 127.0.0.1
   ```

   This IP address will be set to whichever IP address the master server uses.
2. Next, uncomment the line of the MySQL configuration file that contains the `server-id`, which is most probably located at `/etc/my.cn` or `/etc/mysql/mysql.conf.d/mysqld.cnf`.
3. The Unix `sed` command can perform this easily:

   ```
   $ sed -i s/#server-id/server-id/g  /etc/mysql/my.cnf
   ```

   ### Tip

   The `/etc/mysql/my.cnf` string will need to be substituted with the correct filename.
4. Uncomment the line of the MySQL configuration file that contains the `server-id`:

   ```
   $ sed -i s/#log_bin/log_bin/g  /etc/mysql/my.cnf
   ```

   ### Tip

   Again, the `/etc/mysql/my.cnf` string will need to be substituted with the correct filename.
5. Now, MySQL needs to be restarted. You can do this using the following command:

   ```
   $ sudo service mysql restart
   ```
6. The following placeholders should be substituted with the actual values:

   ```
   MYSQLUSER
   ```

MYSQLPASSWORD
MASTERDATABASE
MASTERDATABASEUSER
MASTERDATABASEPASSWORD
SLAVEDATABASE
SLAVEDATABASEUSER
SLAVEDATABASEPASSWORD

Master server set up

The following are the steps for setting up a master server:

1. Grant permission to the slave database user:

```
$ echo  "GRANT REPLICATION SLAVE ON *.* TO 'DATABASEUSER'@'%'
IDENTIFIED BY 'DATABASESLAVEPASSWORD';" | mysql -u MYSQLUSER
-p"MYSQLPASSWORD"
```

2. Next, the permissions must be flushed using the following command:

```
$ echo  "FLUSH PRIVILEGES;" | mysql -u MYSQLUSER -
p"MYSQLPASSWORD"
```

3. Next, switch to the master database using the following command:

```
$ echo  "USE MASTERDATABASE;" | mysql -u MYSQLUSER -
p"DATABASEPASSWORD"
```

4. Next, flush the tables using the following command:

```
$ echo  "FLUSH TABLES WITH READ LOCK;" | mysql -u MYSQLUSER -
p"MYSQLPASSWORD"
```

5. Display the master database status using the following command:

```
$ echo  "SHOW MASTER STATUS;" | mysql -u MYSQLUSER -
p"MYSQLPASSWORD"
```

   Take note of the position and filename from the output:

```
POSITION
FILENAME
```

6. Dump the master database using the following command:

```
$ mysqldump -u root -p"MYSQLPASSWORD"  --opt "MASTERDATABASE"
> dumpfile.sql
```

7. Unlock the tables using the following command:

```
$ echo  "UNLOCK TABLES;" | mysql -u MYSQLUSER -
p"MYSQLPASSWORD"
```

Slave server set up

The following are the steps for setting up a slave server:

1. On the slave server, create the slave database using the following command:

```
$ echo  "CREATE DATABASE SLAVEDATABASE;" | mysql -u MYSQLUSER
-p"MYSQLPASSWORD"
```

2. Import the dump file created from the master database using the following command:

```
$ mysql -u MYSQLUSER -p"MYSQLPASSWORD"  "MASTERDATABASE" <
dumpfile.sql
```

3. Now, the MySQL configuration file uses server-id 2:

```
server-id            = 2
```

4. In the MySQL configuration file, two lines should be uncommented, as follows:

```
#log_bin                        = /var/log/mysql/mysql-
bin.log
expire_logs_days        = 10
max_binlog_size   = 100M
#binlog_do_db           = include_database_name
```

5. You will get the following result:

```
log_bin                 = /var/log/mysql/mysql-bin.log
expire_logs_days        = 10
max_binlog_size    = 100M
binlog_do_db            = include_database_name
```

6. Additionally, the following line needs to be added below `binglog_do_db`:

```
relay-log               = /var/log/mysql/mysql-relay-bin.log
```

7. Now, MySQL needs to be restarted using the following command:

```
$ sudo service mysql restart
```

8. Finally, the master password is set. The master log file and positions are to be set to the filename and the position recorded in step 5. Run the following command:

```
MASTER_PASSWORD='password', MASTER_LOG_FILE='FILENAME',
MASTER_LOG_POS= POSITION;
```

Summary

In this chapter, you learned how to speed up the routing through route caching. You also learned how to replace Laravel entirely with Lumen, the microframework entirely derived from Laravel. Finally, we discussed how Laravel can use a read and write configuration to take full advantage of a master and slave configuration.

Symfony 2.7 was released in May, 2015. It is a long-term support version. This version will be supported for 36 months. Shortly after that, Taylor Otwell made the decision to create the first LTS version of Laravel. This is the first sign that Laravel is firmly positioned in the enterprise space. There is no formal company behind Laravel yet, as there is in the case of Symfony and Zend. Yet there is a large ecosystem of community packages and services such as Laracasts, run by Jeffrey Way who works very closely with Taylor to provide official training videos.

Also, Taylor Otwell runs a service called Envoyer that removes any and all of the initial barriers to Laravel deployment and provides *zero downtime* deployment for Laravel as well as other types of modern PHP projects.

With the arrival of Laravel 5.1 LTS, many new and exciting things will be happening for Laravel. The decision to use many community packages allows Taylor and his community to focus on the most important aspects of the framework without having to reinvent the wheel and maintain many redundant packages. Also, Laravel Collective maintained the packages that have been deprecated—even packages that are eventually removed from Laravel will continue to be supported for many years to come.

In addition to convenient services, such as Envoyer, the next chapter will present a great automation tool that has recently emerged: Elixir.

Chapter 10. Building, Compiling, and Testing with Elixir

In this chapter, the following topics will be covered:

- Installing Node.js, Gulp, and Elixir
- Running Elixir
- Combining CSS and JavaScript files using Elixir
- Setting up notifications
- Running tests with Elixir
- Extending Elixir

Automating Laravel

Throughout the book, many parts of an example application have been built. We discussed the steps involved in creating an application. However, there is more information available about tools to help with scaffolding, boilerplate templates, and building up a RESTful API for a CRUD application. Until recently, not much was written about automating some parts of the development process and deployment process.

A newer area that has emerged in the last few years in the PHP field is the concept of continuous integration and build tools. The popularity of continuous integration and continuous delivery enables teams of developers to constantly release many small improvements to their application many times a day. In this chapter, you will learn how Laravel has a new toolset to empower teams to rapidly and effortlessly deploy versions of their software and build, and combine many of the software's components automatically.

Continuous integration and continuous delivery has caused quite an evolution in the development process, which has drastically changed the way in which software is built. It was not too long ago, however, when the standard deployment process involved simply placing the code on the server. A majority of the early adopters of PHP were simply web designers with a need to add functionality such as a *forum* or *contact us* form. Since most of them were not programmers, most of the practices used in web design and graphic design in general made their way into PHP deployment as well. These practices often involved using an application such as FileZilla to drag-and-drop files from the left-hand side panel (the user's computer) to the right (the server's directories). For the more experienced, performing what was then cryptic UNIX commands using a terminal emulator such as PuTTY.

The insecure file transfer port 21 was used, and everything was sent uncompressed and simply copied to the server. Usually, all of the files were overwritten and often the deployment took almost an hour for a large site with many images and files.

Eventually, source code control systems became pervasive. In the most recent years, SVN and Git have become the industry standard for most software projects. These tools allowed for deployment directly from a code repository.

Most recently, the arrival of composer has created an easy way to simply include entire software packages to add functionality to software applications. The ease with which a developer simply needs to add a single line to the configuration file is exhilarating!

Automating both the development and deployment processes may involve many steps, some of which are listed next.

Deployment

Here are some of the functions of the deployment process:

- Copying certain configuration settings that are relevant to the production environment
- Processing or compiling any **cascading style sheets** (**CSS**) or JavaScript files that were written using a shortcut syntax or preprocessor
- Copying the various assets (source code or images) to mirrors, cluster servers, or content delivery networks
- Modifying the read/write/execute permissions and/or ownership of certain files or directories
- Combining several files into one to reduce the overhead required to perform multiple HTTP calls
- Reducing useless white space and comments (minifying and/or uglifying) in files to reduce their size
- Comparing existing files on the server with the files from our local environment to determine whether or not to overwrite them
- Tagging and/or versioning the source code to allow for possible code rollbacks

Development or deployment

Here are some of the functions of the development or deployment process:

- Verifying that code passed all of the unit, functional, and acceptance tests written to guarantee its quality
- Running scripts that perform various operations
- Performing any migrations, seeding, or other modifications to the database tables
- Obtaining source code control from a hosted source code control system such as GitHub

It is clear that modern development is very complex. An even more difficult aspect of software development is continually recreating the production or final environment while developing.

Towards automation

Tools such as file watchers can run scripts or perform operations every time a file is modified. Additionally, IDEs such as PHPStorm will recognize file extensions and offer to watch the file for changes and allow the developer to perform certain operations. While this approach is acceptable, it is not very portable and each developer would have to create and share a configuration file with the various watchers within the IDE or text editor. This creates a dependency, relying on one single IDE for the entire team.

Also, other approaches such as Bash-shell scripts could be created that run at certain intervals. However, using these scripts requires UNIX-shell coding knowledge. As has been previously demonstrated, tools like artisan help automate many manual tasks. However, most of the default artisan commands were and are still designed to be executed manually.

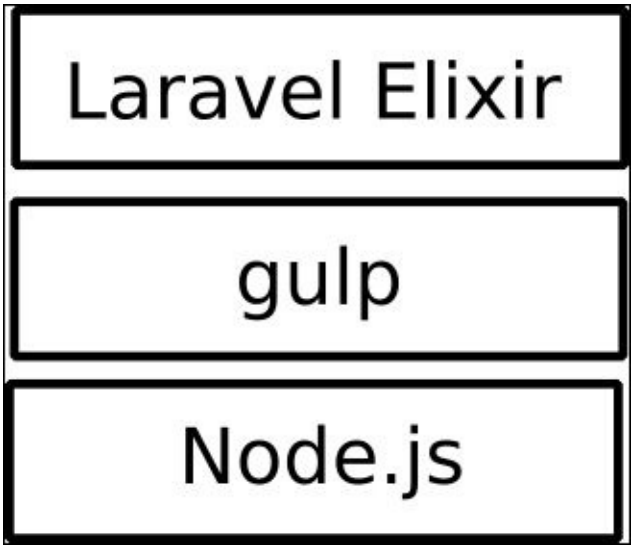
Luckily, two tools that use the Node.js JavaScript platform have emerged: *Grunt* and *gulp*. Both Grunt and gulp have had a considerable amount of success, but gulp has recently become more popular. However, learning to quickly write tasks using gulp is not very easy, especially for a PHP developer who may not be familiar with JavaScript syntax.

Consider the following example code taken from the documentation of gulp:

```
gulp.task('scripts', ['clean'], function() {
  // Minify and copy all JavaScript (except vendor scripts)
  // with sourcemaps all the way down
  return gulp.src(paths.scripts)
    .pipe(sourcemaps.init())
      .pipe(coffee())
      .pipe(uglify())
      .pipe(concat('all.min.js'))
    .pipe(sourcemaps.write())
    .pipe(gulp.dest('build/js'));
});
```

From Gulp to Elixir

Luckily, the Laravel community is always forward thinking and focuses on reducing complexity. An official community tool named **Elixir** has emerged to facilitate the use of gulp. Gulp is built on top of Node.js, and Elixir is built on top of gulp, creating a wrapper:



Note

Laravel Elixir is not to be confused with the dynamic, functional language with the same name. The other Elixir uses the Erlang virtual machine, while Laravel Elixir uses gulp and Node.js

Getting started

The first step is to install Node.js if it is not already installed on the development computer.

Note

Instructions can be found at the following URL:

https://nodejs.org

Installing Node.js

With a Debian-based operative system such as Ubuntu, installing Node.js could be as simple as using the `apt` package manager. From the command line, use the following command:

```
$ sudo apt-get install -y nodejs
```

Please refer to the installation instructions for the correct operating system from the Node.js website (https://nodejs.org).

Installing the Node.js package manager

The next step involves installing gulp, which Elixir will use to run its tasks. For this step, the **Node.js package manager** (**npm**) is required. If `npm` is not already installed, then the `apt` package installer is to be use. The following command will be used to install `npm`:

```
$ sudo apt-get install npm
```

The npm uses a `json` file to manage the project's dependencies: `package.json`. This file is found in the root of the Laravel project directory and has the following format:

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

Gulp and Laravel Elixir are installed as dependencies.

Installing Gulp

The following command is used to install `gulp`:

```
$ sud onpm install --global gulp
```

Installing Elixir

Once Node.js, npm, and gulp are installed, the next step is to install Laravel Elixir. By running `npm` install without any arguments, `npm` will read its configuration file and Laravel Elixir will be installed:

```
$ npm install
```

Running Elixir

By default, Laravel contains a `gulpfile.js` file that is used by gulp to run its tasks. The file contains a `require` method that includes everything needed for the tasks to run:

```
var elixir = require('laravel-elixir');

/*
 |--------------------------------------------------------------------
-
 | Elixir Asset Management
 |--------------------------------------------------------------------
-
 |
 | Elixir provides a clean, fluent API for defining some basic
gulp tasks
 | for your Laravel application. By default, we are compiling the
Sass
 | file for our application, as well as publishing vendor
resources.
 |
 */

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

The first mix is shown as an example: `app.less`. To run gulp, simply type `gulp` at the command line as follows:

```
$  gulp
```

The output is shown here:

```
[21:23:38] Using gulpfile /var/www/laravel.example/gulpfile.js
[21:23:38] Starting 'default'...
[21:23:38] Starting 'less'...
[21:23:38] Running Less: resources/assets/less/app.less
[21:23:41] Finished 'default' after 2.35 s
[21:23:43] gulp-notify: [Laravel Elixir] Less Compiled!
[21:23:43] Finished 'less' after 4.27 s
```

The first line indicates that the gulp file was loaded. The following lines show each task running. The `less` job deals with the cascading style sheet preprocessor

Less.

Setting up notifications

If your development environment is a Vagrant Box, installing `vagrant-notify` will allow Laravel Elixir to interact directly with the host and display native messages directly in the operating system. To install it, the following command should be run from the host operating system:

```
$ vagrant plugin install vagrant-notify
```

Here's the screenshot of a notification showing that PHPUnit tests have failed:



The installation instructions depend on each operating system.

Note

For more information, visit https://github.com/fgrehm/vagrant-notify.

Combining CSS and JavaScript files with Elixir

Possibly, the single most important step of the deployment process is to combine and minify CSS and JavaScript files. Minifying and combining five JavaScript files and three CSS files means that instead of eight HTTP requests, there will be only one. Also, by minifying the files by removing white space, line breaks, comments, and other techniques, such as shortening variable names, the file size will be reduced to a fraction of its original size. Despite the advantage, there are still many websites that continue to use unminified and uncombined CSS and JavaScript files.

Elixir provides a simple means to easily combine and minify files. The following code illustrates this example:

```
elixir(function(mix) {
    mix.scripts().styles();
});
```

The two methods, `scripts()` and `styles()`, will combine all of the JavaScript and CSS files into single files, `all.js` and `all.css`, respectively. By default, the two functions expect the files to be located at `/resources/assets/js` and `/resources/assets/css`.

When the gulp command has finished, the output will be like this:

```
[00:36:20] Using gulpfile /var/www/laravel.example/gulpfile.js
[00:36:20] Starting 'default'...
[00:36:20] Starting 'scripts'...
[00:36:20] Merging: resources/assets/js/**/*.js
[00:36:20] Finished 'default' after 246 ms
[00:36:20] Finished 'scripts' after 280 ms
[00:36:20] Starting 'styles'...
[00:36:20] Merging: resources/assets/css/**/*.css
[00:36:21] Finished 'styles' after 191 ms
```

Notice how the output conveniently states which directories were scanned. The contents are combined, referred to as merged in this context, but not minified. This is because, during development, debugging is too difficult on minified files. If only a certain file is to be merged, then the name of the file may be passed into the function as the first parameter:

```
mix.scripts('app.js');
```

If multiple files are to be merged, then an array of filenames may be passed into the function as a first parameter:

```
mix.scripts(['app.js','lib.js']);
```

In a production environment, having minified files is desirable. To have Elixir minify both the CSS and JavaScript, add the `--production` option to the gulp command as follows:

```
$ gulp --production
```

This will produce the desired minified output. The default output directory is located at:

```
/public/js
/public/css
```

Compiling with Laravel Elixir

Laravel Elixir is great at performing routine tasks that would otherwise require learning scripting languages. The following sections will demonstrate each of the various types of compiling that Elixir can perform.

Compiling Sass and Less

Cascading style sheet preprocessors `Less` and `Sass` emerged from the need to enhance the capabilities of CSS. It does not contain any variables for example. `Less` and `Sass` allow frontend developers take advantage of variables and other features that are syntactically familiar. The following code is an example of standard CSS. The DOM elements `p` and `li` (paragraph and list item respectively), and any element with a `post` class will have a `font-family` Arial, with sans-serif as a fallback and black as its color:

```
p, li, .post {
  font-family: Arial, sans-serif;
  color: #000;
}
```

Next, using the `Sass` CSS pre-processor, both the font-family and text color will be substituted with two variables: `$text-font` and `$text-color`. This allows easy maintenance when changes are needed. Also, the variables may be shared. The code is as follows:

```
$text-font:    Arial, sans-serif;
$text-color: #000;

p, li, .post {
  font: 100% $text-font;
  color: $text-color;
}
h2 {
  font: 2em $text-font;
  color: $text-color;
}
```

The `Less` preprocessor uses `@` instead of `$`; therefore, its syntax looks more like an annotation than a `php` variable:

```
@text-font:    Arial, sans-serif;
@text-color: #000;

p, li, .post {
  font: 100% @text-font;
  color: @text-color;
}
h2 {
  font: 2em @text-font;
  color: @text-color;
```

```
}
```

There is an extra step that needs to be performed, since it will not be interpreted by a browser engine. The added step is to compile the `Less` or `Sass` code into real CSS. This introduces extra time in the development phase; thus, Elixir helps by automating the process.

In the previous Laravel Elixir example, the `less` function took only the filename, `app.less`, as its sole argument. Now, the example should be a bit clearer. Also, `less` may take an array of arguments that will be compiled.

The `less` method searches in `/resources/assets/less` and the output will be placed in `public/css/` by default:

```
elixir(function(mix) {
    mix.less([
        'style.less',
        'style-rtl.less'
    ]);
});
```

Compiling CoffeeScript

CoffeeScript is a programming language that compiles into JavaScript. Like Less and Sass, its goal is to simplify or extend the functionality of the language that it compiles to. In the case of CoffeeScript, it simplifies Javascript by requiring less keystrokes. In the following JavaScript code, two variables—an array and an object—are created:

```
var available, list, room;

room = 14;

available = true;

list = [101,102,311,421];

room = {
  id: 1,
  number: 102,
  status: "available"
}
```

In the following CoffeeScript code, the syntax is very similar, but there is no semicolon required and there is no need for `var` to create a variable. Also, indentation is used to define the object's attributes. The code is as follows:

```
room = 14

available = true

list = [101,102,311,421]

room =
  id: 1
  number: 102
  status: "available"
```

In this CoffeeScript example, there are only a few less characters; however, for a programmer, less keystrokes can help increase speed and efficiency. To add the coffee compiler to Elixir, simply use the `coffee` function, as shown in the following code:

```
elixir(function(mix) {
    mix.coffee([
        'app.coffee'
```

```
    ]);
});
```

A summary of compiler commands

The following table shows the mapping between preprocessor, language, function, and where each function expects the source to be. The last column on the right shows the directory and/or name of the resultant combined file.

processor	Language	function	Source directory
Less	CSS	`less()`	`/resources/assets/less/file(s).less`
Sass	CSS	`sass()`	`/resources/assets/sass/file(s).scss`
N/A	CSS	`styles()`	`/resources/assets/css/`
N/A	JavaScript	`scripts()`	`/resources/assets/js/`
CoffeeScript	JavaScript	`coffee()`	`/resources/assets/coffee/`

Saving with a different name

Optionally, each method takes a second parameter that will override the default location. To use a different directory, (in this case, a directory called `app`), simply add the directory as a second parameter:

```
mix.scripts(null,'public/app/js').styles(null,'public/app/css');
```

In this example, the files will be saved at `public/app/jsand public/app/css`.

Putting everything together

Finally, let's put everything together to draw an interesting conclusion. Since CoffeeScript scripts and `less` and `sass` files are not merged but copied into the destination directly, we first save the CoffeeScript, `less`, and `sass` files into the directories where Elixir expects the JavaScript and CSS files to be. Then, we instruct Elixir to merge and minify all of the JavaScript and CSS files into two merged and minified files. The code is as follows:

```
elixir(function(mix) {
    mix.coffee(null,'resources/assets/js')
        .sass(null,'resources/assets/css')
        .less(null,'resources/assets/css')
        .scripts()
        .styles();
});
```

Tip

It is extremely important to remember that Elixir overwrites files without verifying that the file exists, so a unique name would need to be chosen for each of the files. When the command is finished, `all.js` and `all.css` will be merged and minified in the `public/js` and `public/css` directories.

Running tests with Elixir

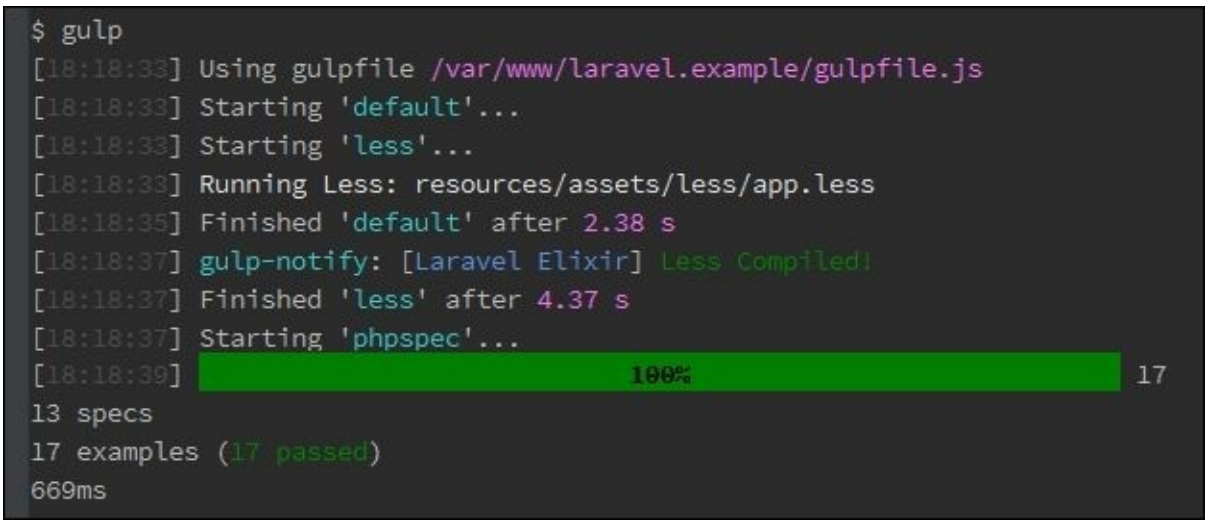
In addition to compiling and sending notifications, Elixir may also be used to automate the launching of tests. The following sections will discuss how Elixir can be used for both PHPSpec and PHPUnit.

PHPSpec

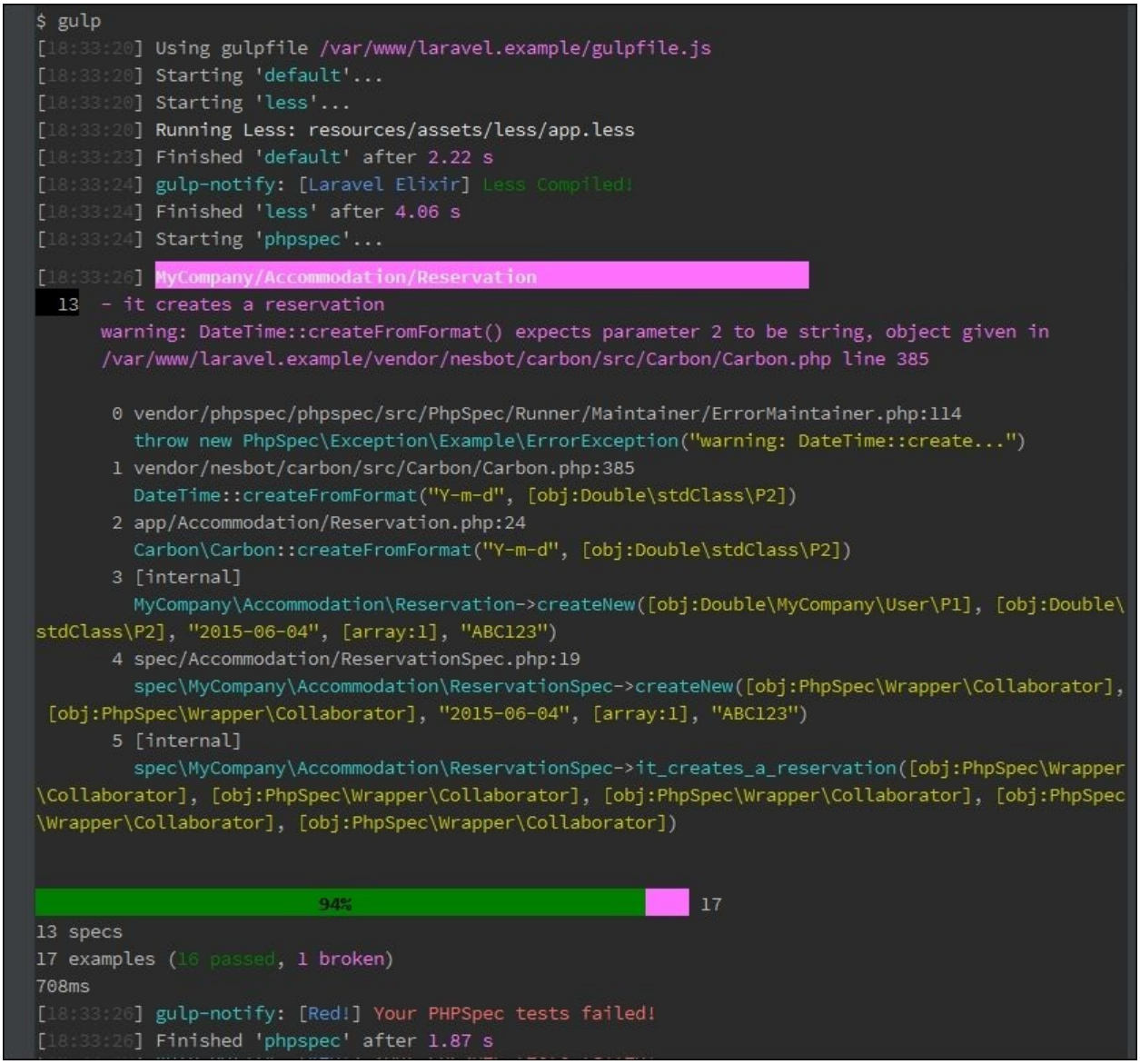
The first step would be to run the PHPSpec tests to automate code testing. By adding `phpSpec()` to our `gulpfile.js` as follows, PHPSpec tests will run:

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

The output is shown in the following screenshot. The PHPSpec output is maintained, so the test output is very useful:



When the PHPSpec tests fail, the results are easily readable:



A screenshot of Laravel Elixir's output

In this example, phpspec encountered an error in the **it creates a reservation test** line as shown in the preceding screenshot.

PHPUnit

Similarly, we may add PHPUnit to our suite of tests by adding `phpUnit` to the list of tasks as follows:

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

Creating custom tasks

Elixir gives us the ability to create custom tasks to do virtually anything. One example of a custom task that we could write is to scan the controllers for annotations. All custom tasks require `gulp` and `laravel-elixir`. It is important to remember that the programming language used is JavaScript, so the syntax may or may not be familiar, but it is easy enough to learn quickly. If commands will be executed from the command-line interface, then we shall also import gulp-shell. The code is as follows:

```
var gulp = require('gulp');
var elixir = require('laravel-elixir');
var shell = require('gulp-shell');


/*
 |--------------------------------------------------------------------
-
 | Route Annotation Scanner
 |--------------------------------------------------------------------
-
 |
 | We'll run route:scan Artisan to scan for changed files.
 | Output is written to storage/framework/routes.scanned.php
 |
*/

 elixir.extend('routeScanning', function() {
                 gulp.task('routeScanning', function() {
                         return gulp.src('').
      pipe(shell('php artisan route:scan'));
                 });

     return this.queueTask('routeScanning');
 });
```

In this code, we first extend Elixir and give the method a name, for example, `routeScanning`. Then, a gulp task is defined and the first argument to the `task` method is the name of the command. The second command is a closure containing the code that will be executed and returned.

Finally, the task is queued for execution by passing the name of the command into the `queueTask` method.

Add this script to our chain as follows:

```
elixir(function(mix) {
    mix.routeScanning();
});
```

The output will be like this:

```
$ gulp
[23:24:19] Using gulpfile /var/www/laravel.example/gulpfile.js
[23:24:19] Starting 'default'...
[23:24:19] Starting 'routeScanning'...
[23:24:19] Finished 'default' after 12 ms
[23:24:20] Finished 'routeScanning' after 1 s
```

Since the `pipe` function allows chaining of commands, it is easy to add in a notification that will alert the notification system, as follows:

```
var gulp = require('gulp');
var elixir = require('laravel-elixir');
var shell = require('gulp-shell');
var Notification = require('./commands/Notification');

 elixir.extend('routeScanning', function() {
                 gulp.task('routeScanning', function() {
                         return gulp.src('').
                             pipe(shell('php artisan
route:scan')).
                             pipe(new
Notification().message('Annotations scanned.'));
                 });
     return this.queueTask('routeScanning');

 });
```

Here, the `Notification` class is pulled in and a new notification is created to send the message `Annotations scanned.` to the notification system.

Running the code produces the following output. Notice that the `gulp-notify` has been added:

```
$ gulp
[23:46:59] Using gulpfile /var/www/laravel.example/gulpfile.js
[23:46:59] Starting 'default'...
[23:46:59] Starting 'routeScanning'...
[23:46:59] Finished 'default' after 38 ms
PHP Warning:  Module 'xdebug' already loaded in Unknown on line 0
```

```
Routes scanned!
[23:47:00] gulp-notify: [Laravel Elixir] Annotations scanned
[23:47:00] Finished 'routeScanning' after 1.36 s
```

Setting up a file watcher

Obviously, running gulp every time we want to compile the cascading style sheets or scan for annotations is tedious. Fortunately, a watcher mechanism is built into Elixir. To invoke it, simply run the following command:

```
$ gulp watch
```

This will allow any tasks that have been placed into the `gulpfile.js` chain to automatically run when certain changes occur. The code necessary to enable this in the annotation task is as follows:

```
 this.registerWatcher("routeScanning",
"app/Http/Controllers/**/*.php");
```

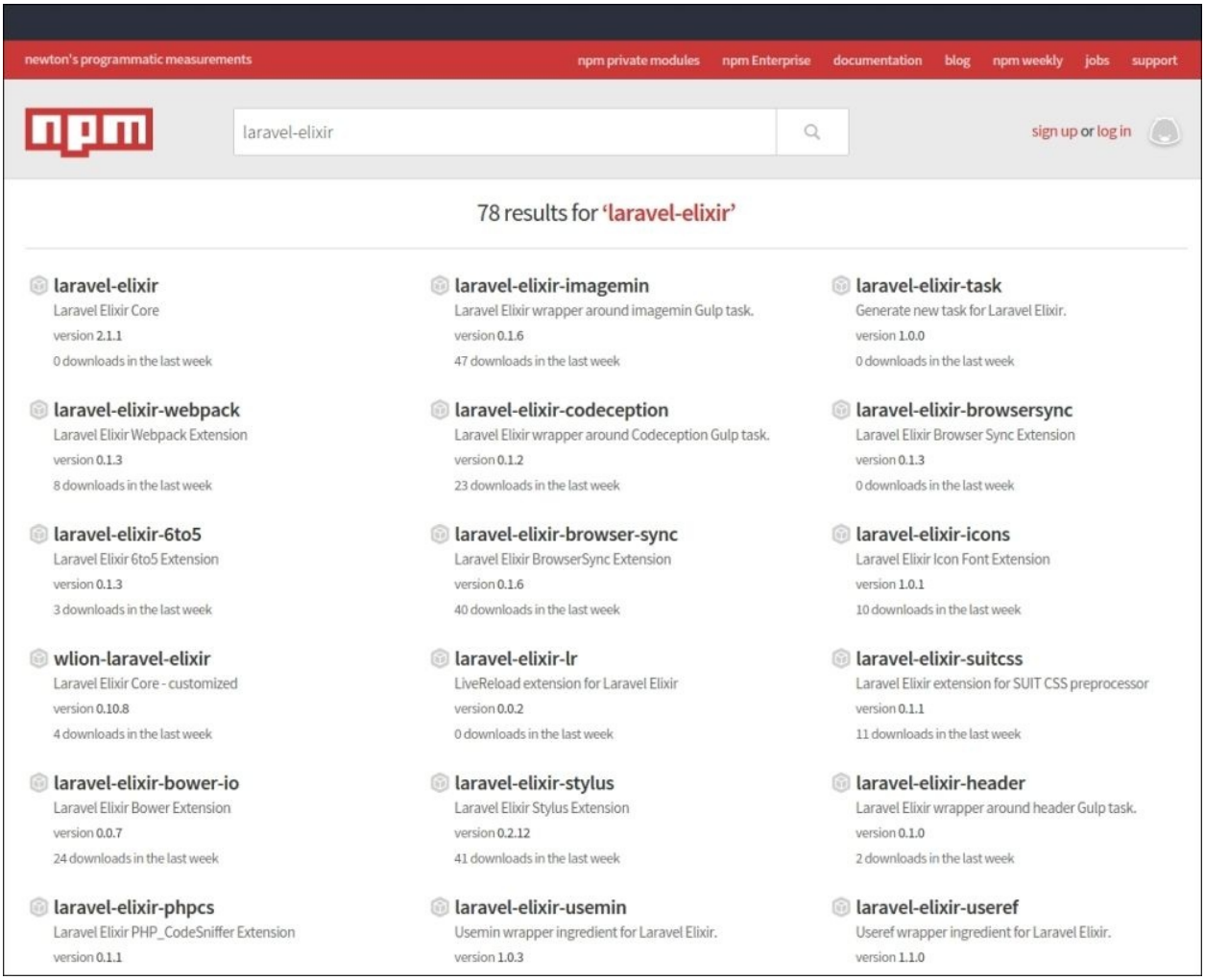
The preceding code registers a watch. The first argument is the `routeScanning` task. The second command is the directory pattern that will be watched for modifications.

Since we know that route annotations will be inside the controllers, we may set the path to look only inside the `app/Http/Controllers/` directory. The regex style syntax will match on any file with a `php` extension in any one of the directories that are located under controllers.

Now, whenever a file matching the pattern is modified, the `routeScanning` task, as well as any other tasks that are watching files that match the same pattern.

Additional Laravel Elixir tasks

The npm website provides more than 75 tasks that deal with testing, JavaScript, CSS, and more. The `npm` website is located at http://npmjs.com.



A screenshot of the npm website containing many useful Laravel Elixir tasks

Summary

In this chapter, you learned how Elixir's growing task list can help both the full-stack developer as well as a team of developers. Some of the tasks are related to frontend development, such as compiling, combining, and minifying CSS and JavaScript. Other tasks are related to backend development, such as behavior-driven development. Integrating these tasks into the daily development workflow will make it easy for the entire team to understand the steps that are necessary in the continuous integration server, where Elixir will execute its tasks, such as testing and compiling, to prepare the files for development into production.

Since Elixir is built upon gulp, the future of the Elixir will continue to be enriched as both the gulp and Elixir communities continue to grow and new contributors continue contributing to Elixir.

Index

A

- amenitiable relationship
  - about / Amenitiable relationships
  - amenity table structure / The Amenity table structure
  - Amenity model / The Amenity model
  - Accommodation model / The Accommodation model
  - Room model / The Room model
- annotations
  - about / Annotations in other programming languages, Additional annotations, Other annotations
  - in Java / Annotations in Java
  - in C# / Annotations in C#
  - in PHP / Annotations in PHP
  - HTTP verbs / HTTP verbs
  - @Any / Other annotations
  - @Controller / Other annotations
  - @Middleware / Other annotations
  - @Route / Other annotations
  - @Where / Other annotations
  - @Resource / Other annotations
- Authenticate class
  - about / Default middleware – the Authenticate class
- automatic scanning
  - about / Automatic scanning

B

- basic middleware structure
  - about / The basic middleware structure
- basic operations
  - about / Basic operations
  - find() method / Finding one
  - where method / The where method
  - chaining functions / Chaining functions
  - all() method / Finding all
- Behat
  - about / Functional testing with Behat
  - functional testing with / Functional testing with Behat
- behavior driven development (BDD)
  - about / phpspec

C

- command bus
  - about / The command bus
- contracts
  - about / Contracts
- controller
  - creating / Controller creation
- controllers
  - about / Controllers
  - creating / Controllers
- CRUD
  - essentials / Essential CRUD
- CRUD(L)
  - read / cRudl – read
  - list / crudL – list
  - create / Crudl – create
  - update / crUdl – update
  - delete / cruDl – delete
- custom middleware
  - about / Custom middleware – Log
  - Log model / The Log model
  - Log model migration / Log model migration
- custom tasks
  - creating / Creating custom tasks

D

- database testing
  - with PHPUnit / Database testing with PHPUnit
- Data Transfer Object (DTO) / Request routing
- Docblock annotations
  - in Laravel / DocBlock annotations in Laravel
- DocBlock annotations
  - about / Laravel
  - using, in Laravel 5 / Using annotations in Laravel 5
  - using / Advantages
- Doctrine / Annotations in PHP
- Domain-driven Design (DDD) / Request routing

E

- Elixir
  - about / From Gulp to Elixir
  - installing / Installing Elixir
  - running / Running Elixir
  - CSS and JavaScript files, combining with / Combining CSS and JavaScript files with Elixir
  - tests, running with / Running tests with Elixir
  - using, for PHPSpec / PHPSpec
  - using, for PHPUnit / PHPUnit
- Eloquent model casting / Eloquent model casting
- Eloquent relations
  - about / Eloquent relations
  - one-to-one / One-to-one
  - one-to-many / One-to-many
  - many-to-many / Many-to-many
  - has-many-through / Has-many-through
- entity creation
  - about / Entity creation
- example, form builder
  - about / Our example
  - selectMonth method / Month select
  - date select, selectRange method used / Date select
  - year select, selectRange method used / Year select
  - form macros / Form macros

F

- FastCGI Process Manager (FPM) / Towards the enterprise
- file watcher
  - setting up / Setting up a file watcher
- form builder
  - history / History
  - example / Our example
  - conclusion / Conclusion
- Framework Interoperability Group (PHP-FIG)
  - about / PSR
  - URL / PSR
- functional testing
  - with Behat / Functional testing with Behat

G

- Gulp
  - about / From Gulp to Elixir
  - installing / Installing Gulp

H

- handle
  - about / Handle
- Homestead
  - about / Installing and configuring Laravel
  - URL / Installing and configuring Laravel
- HTML package
  - installing / Installing the HTML package
- HTTP kernel
  - about / The HTTP kernel
- HTTP verbs
  - about / HTTP verbs
  - @Delete / HTTP verbs
  - @Get / HTTP verbs
  - @Options / HTTP verbs
  - @Patch / HTTP verbs
  - @Post / HTTP verbs
  - @Put / HTTP verbs

L

- Laravel
  - evolution / A new era
  - installing / Installing and configuring Laravel, Installation
  - URL / Installing and configuring Laravel
  - configuration / Configuration
  - namespacing / Namespacing
  - syntax / Laravel
  - enterprise / Towards the enterprise
  - automating / Automating Laravel, Towards automation
  - deployment process / Deployment
  - development or deployment process / Development or deployment
- Laravel 5 master template
  - about / The master template
  - features / The master template
- Laravel Elixir
  - about / Compiling with Laravel Elixir
  - additional tasks / Additional Laravel Elixir tasks
- Laravel Elixir, compiling with
  - Sass and Less, compiling / Compiling Sass and Less
  - CoffeeScript, compiling / Compiling CoffeeScript
  - compiler commands / A summary of compiler commands
  - saving, with different name / A summary of compiler commands
- leaner app
  - about / A leaner app
- LogMiddleware class / Custom middleware – Log
- long-term support (LTS) / A new era
- Lumen / Towards the enterprise

M

- master/slave database configuration
  - creating / Creating a master/slave database configuration
  - master server, setting up / Master server set up
  - slave server, setting up / Slave server set up
- middleware
  - using / Using middleware
  - exclusion / Middleware exclusion and inclusion
  - inclusion / Middleware exclusion and inclusion
  - conclusion / Conclusion
- middleware, using
  - route groups / Route groups
  - multiple middleware, in route groups / Multiple middleware in route groups
- migration
  - about / Using Laravel's migration feature
  - operations / Using Laravel's migration feature
  - example / An example of migration
  - table, creating / Creating the table
  - $table->timestamps() / $table->timestamps();
  - generating / Generating the migrations
- migration, creating from schema
  - about / From schema to migration
  - require-dev command / Composer's require-dev command
  - Laravel providers / Laravel's providers array
  - composer update command / The composer update command
- migration anatomy
  - about / Migration anatomy
  - list tables / List tables
  - softDeletes / The softDelete and timestamp properties
  - timestamps / The softDelete and timestamp properties
- model binding
  - about / Model binding
  - read revisited / Read revisited
  - list revisited / List revisited
  - update revisited / Update revisited
  - delete revisited / Delete revisited
- MyCompany database schema

- about / The MyCompany database schema

N

- nested controllers
  - about / Nested controllers
  - Accommodation hasMany rooms / Accommodation hasMany rooms
  - room belongsTo accommodation / Room belongsTo accommodation
  - Eloquent relations / Eloquent relations
  - nested update / Nested update
  - nested create / Nested create
- nested relation / Eloquent relations
- Node.js
  - URL / Getting started
  - installing / Installing Node.js
- Node.js package manager (npm)
  - installing / Installing the Node.js package manager
- notifications
  - setting up / Setting up notifications

O

- Object-relational mapping (ORM) / $table->timestamps();
- operations, migration
  - install / Using Laravel's migration feature
  - refresh / Using Laravel's migration feature
  - reset / Using Laravel's migration feature
  - rollback / Using Laravel's migration feature
  - status / Using Laravel's migration feature

P

- Phansible
  - about / Installing and configuring Laravel
  - URL / Installing and configuring Laravel
- phpspec
  - about / phpspec
  - designing with / Designing with phpspec
  - specifying with / Specifying with phpspec
  - rules of test-driven development / Red, green, refactor
  - cleaning / Tidying things up
- PHPSpec
  - about / PHPSpec
- PHP Standard Recommendation (PSR)
  - about / PSR
- PHPUnit
  - about / Database testing with PHPUnit, PHPUnit
  - database testing with / Database testing with PHPUnit
  - running / Running PHPUnit
- polymorphic relations
  - about / Polymorphic relations
  - amenitiable relationship / Amenitiable relationships
  - many-to-many polymorphic relations / Many-to-many polymorphic relations
  - has relationships / Has relationships
  - eager loading / Eager loading
- PuPHPet
  - URL / Installing and configuring Laravel
  - about / Installing and configuring Laravel

R

- recycle bin / $table->timestamps();
- request routing
  - about / Request routing
  - user stories / User stories
  - queued event handlers / Queued event handlers
  - waiting list command / The waiting list command
  - queued commands / The queued commands
  - console commands / The console command
  - command scheduler / The command scheduler
- resource controller, using DocBlock annotations
  - about / Resource controller using DocBlock annotations
- RESTful APIs
  - about / RESTful APIs in Laravel
  - bonus features / Bonus features
- RoomWasReserved event class / Command to event
- route-based middleware
  - about / Route middleware unravelled
  - default middleware / Default middleware – the Authenticate class
  - contracts / Contracts
  - handle / Handle
  - custom middleware / Custom middleware – Log
  - terminable middleware / Terminable middleware
- route caching
  - about / Route caching, Route caching
  - illuminate routing / Illuminate routing
  - Lumen / Lumen
  - Laravel and Lumen, comparing / Comparison between Laravel and Lumen
  - Lean application development / Lean application development
  - read/write / Read/write
  - master table / Master table
  - slave table / Slave table
  - read/write, configuring / Configuring read/write
- routes
  - scanning / Scanning routes

S

- scalability issues
  - about / Scalability issues
- seeds
  - creating / Creating seeds
- single method routing
  - about / Single method routing
- static HTML to static methods
  - form tag / The form tag
  - text input field / The text input field
  - label tag / The label tag
  - checkbox / Checkbox
  - submit button / The submit button
  - anchor tag with links / The anchor tag with links
  - form, closing / Closing the form
  - resultant form / The resultant form
- Symfony
  - syntax / Symfony

T

- terminable middleware
  - about / Terminable middleware
  - logging, as terminable / Logging as terminable
- test-driven development
  - phpspec / phpspec
- tests
  - running, with Elixir / Running tests with Elixir

U

- user stories
  - about / User stories
  - for coding / User stories to code
  - room, searching / Searching for the room
  - controller to command / Controller to command
  - command to event / Command to event
  - ReserveRoomCommandHandler class / The ReserveRoomCommandHandler class
  - event handler / Event to handler

V

- Vagrant
  - about / Installing and configuring Laravel

W

- web pages, building with Laravel
  - about / Building web pages with Laravel
  - master template / The master template
  - example page / An example page
  - from static HTML to static methods / From static HTML to static methods

Z

- Zend
  - syntax / Zend
- Zend Framework (ZF) / Annotations in PHP