Community Experience Distilled

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

Mastering Laravel

Christopher John Pecoraro

What you will learn from this book

- Convert specifications into classes and functions, using a specification-based behavioral-driven design (BDD)
- Create a deployment script for a continuous delivery environment
- Use Eloquent to interact with the database in an object-oriented manner
- Create RESTful APIs to allow an application to interact with other programs
- Easily scale software with route caching and the read/write configuration
- Use DocBlock annotations to enhance controllers and reduce the amount of code required

Who this book is written for

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

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

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Designing Done Right with phpspec'
- A synopsis of the book’s content
- More information on Mastering Laravel
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.
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.
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 namespaces 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.
Chapter 1

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:

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

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

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

use PhpSpec\ObjectBehavior;
use Prophecy\Argument;

class AccommodationRepositorySpec extends ObjectBehavior
{
    function it_is_initializable()
    {
```
Designing Done Right with phpspec

```php
<?php
namespace MyCompany;

class AccommodationRepository
{
}
```

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\AccommodationRepository 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
<?php
namespace MyCompany;

class AccommodationRepository
{
}
```

---

**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:

```bash
# 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:

```bash
# 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
<?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:

```php
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:

```php
$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:

```php
if ( $end->diffInDays($start) < 1 ) {
```
The preceding code isn't quite readable. We can refactor it in the following way:

```php
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:

```php
const MINIMUM_STAY_LENGTH = 1;
```

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

```php
/**
 * @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:

```php
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:

```php
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];
    $this->shouldThrow('InvalidArgument Exception')->duringCreateNew($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:

```php
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:

```php
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.');</n}
```

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:

```php
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:

```php
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:

```php
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:

```php
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:

```php
$rooms = [$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
<?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, $start)) {
            throw new \InvalidArgumentException
                ('Requires a stay length from ' . self::MINIMUM_STAY_LENGTH . ' to ' .
                 self::MAXIMUM_STAY_LENGTH . ' days.');</n        }
        if (!is_array($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.');</n        }

        return $this;
    }
}
```
private function daysAreLessThanMinimumAllowed($end, $start)
{
    return $end->diffInDays($start) <
        self::MINIMUM_STAY_LENGTH;
}

private function daysAreGreaterThanMaximumAllowed($end, $start)
{
    return $end->diffInDays($start) >
        self::MAXIMUM_STAY_LENGTH;
}

private function daysAreWithinAcceptableRange($end, $start)
{
    if ($this->daysAreLessThanMinimumAllowed($end, $start)
        || $this->daysAreGreater ThanMaximumAllowed($end, $start)) {
        return false;
    } else {
        return true;
    }
Designing Done Right with phpspec

```php
/**
 * @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:

```bash
$ php artisan make:controller RoomController
```

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

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

use Illuminate\Http\Request;
class RoomController extends Controller {
```
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:command ReserveRoomCommand --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
<?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.
     *
     * @param $user
     * @param $start_date
     * @param $end_date
     * @param $rooms
     */
    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
<?php namespace MyCompany\Handlers\Commands;

use MyCompany\Commands\ReserveRoomCommand;

use Illuminate\Queue\InteractsWithQueue;

class ReserveRoomCommandHandler {

    /**
     * Create the command handler.
     *
     * @return void
     */
```
We will fill in the handle method with the validation of the reservation, as follows:

```php
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.
Where to buy this book
You can buy Mastering Laravel from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.