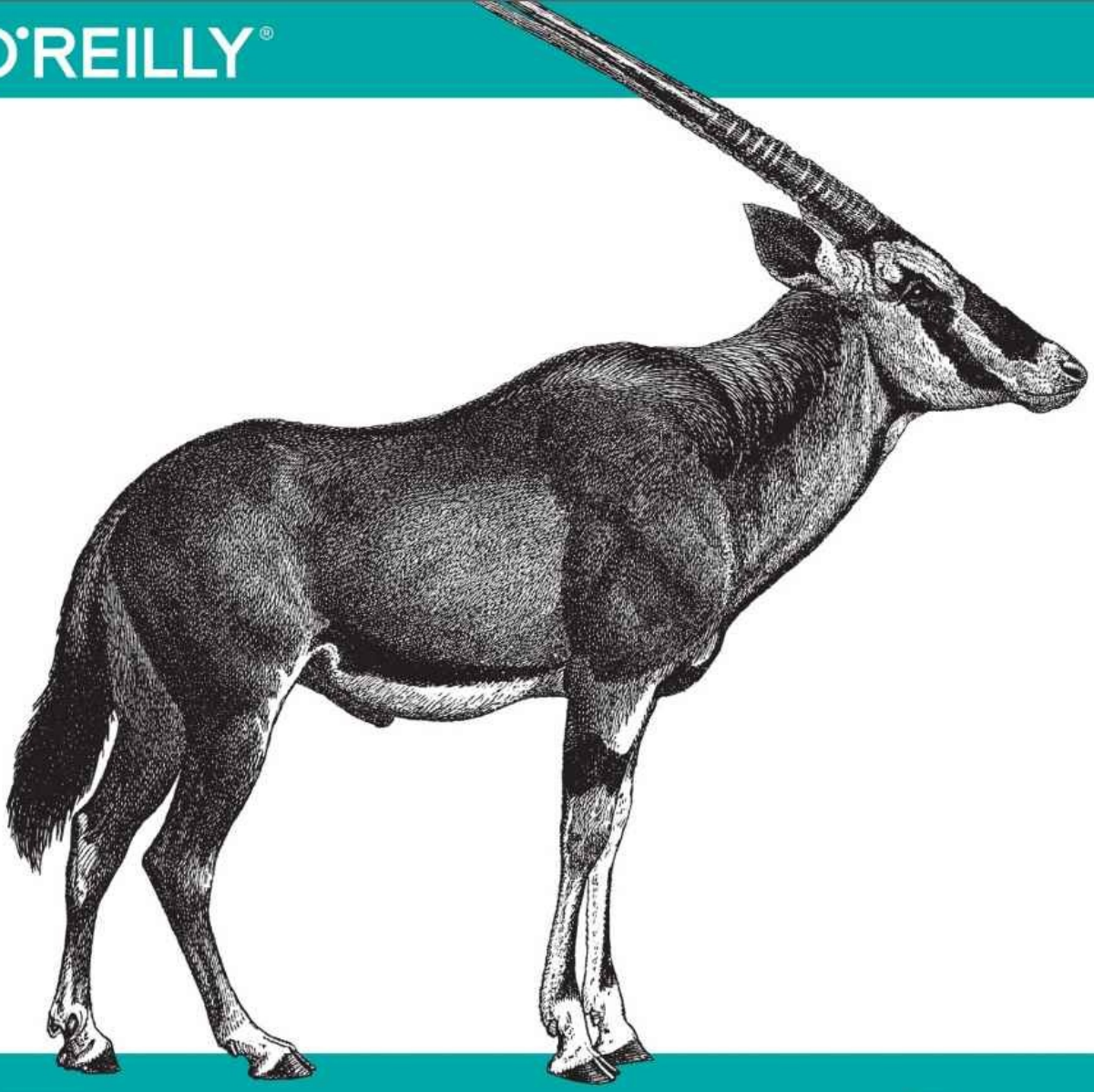
O'REILLY®



# Laravel
# Up & Running

A FRAMEWORK FOR BUILDING MODERN PHP APPS

Matt Stauffer

# Laravel: Up and Running

A Framework for Building Modern PHP Apps

**Matt Stauffer**

**Laravel: Up and Running**

by Matt Stauffer



Printed in the United States of America.

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com/safari). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

- Editor: Allyson MacDonald
- Production Editor: Colleen Lobner
- Copyeditor: Rachel Head
- Proofreader: Kim Cofer
- Indexer: Angela Howard
- Interior Designer: David Futato
- Cover Designer: Randy Comer
- Illustrator: Rebecca Demarest
- December 2016: First Edition

**Revision History for the First Edition**

- 2016-11-14: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781491936085 for release details.



978-1-491-93608-5

[LSI]

**Dedication**

This book is dedicated to my gracious and inspiring wife, Tereva, my joyful and courageous son, Malachi, and my beautiful daughter, Mia, who spent the majority of this book's creation in her mama's belly.

# Preface

The story of how I got started with Laravel is a common one: I had written PHP for years, but I was on my way out the door, pursuing the power of Rails and other modern web frameworks. Rails in particular had a lively community, a perfect combination of opinionated defaults and flexibility, and the power of Ruby Gems to leverage prepackaged common code.

Something kept me from jumping ship, and I was glad for that when I found Laravel. It offered everything I was drawn to in Rails, but it wasn't just a Rails clone; this was an innovative framework with incredible documentation, a welcoming community, and clear influences from many languages and frameworks.

Since that day I've been able to share my journey of learning Laravel through blogging and speaking at conferences; I've written dozens of apps in Laravel for side and work projects, and I've met thousands of Laravel developers online and in person. I have plenty of tools in my toolkit at our consultancy, but I am honestly happiest when I sit down in front of a command line and type `laravel new project`.

# What This Book Is About

This is not the first book about Laravel, and it won't be the last. I don't intend for this to be a book that covers every line of code or every implementation pattern. I don't want this to be the sort of book that goes out of date when a new version of Laravel is released. Instead, its primary purpose is to provide developers with a high-level overview and concrete examples to learn what they need to get started, as quickly as possible. Rather than mirroring the docs, I want to help you understand the foundational concepts behind Laravel.

Laravel is a powerful and flexible PHP framework. It has a thriving community and a wide ecosystem of tools, and as a result it's growing in appeal and reach. This book is for developers who already know how to make websites and applications and want to quickly learn how to do so in Laravel.

Laravel's documentation is thorough and excellent. If you find that I don't cover any particular topic deeply enough for your liking, I encourage you to visit the online documentation and dig deeper into that particular topic.

I think you will find the book a comfortable balance between high-level introduction and concrete application, and by the end you should feel comfortable writing an entire application in Laravel, from scratch. And, if I did my job well, you'll be excited to try.

# Who This Book Is For

This book assumes knowledge of basic object-oriented programming practices, PHP (or at least the general syntax of C-family languages), and the basic concepts of the Model–View–Controller (MVC) pattern and templating. If you've never made a website before, you may find yourself in over your head. But as long as you have some programming experience, you don't have to know anything about Laravel before you read this book — we'll cover everything you need to know, from the simplest "Hello, world!"

Laravel can run on any operating system, but there will be some Bash (shell) commands in the book that are easiest to run on Linux/Mac OS. Windows users may have a harder time with these commands and with modern PHP development, but if you follow the instructions to get Homestead (a Linux virtual machine) running, you'll be able to run all of the commands from there.

# How This Book Is Structured

This book is structured in what I imagine to be a chronological order: if you're building your first web app with Laravel, the early chapters cover the foundational components you'll need to get started, and the later chapters cover less foundational or more esoteric features.

Each section of the book can be read on its own, but for someone new to the framework, I've tried to structure the chapters so that it's actually very reasonable to start from the beginning and read until the end.

Where applicable, each chapter will end with two sections: "Testing" and "TL;DR." If you're not familiar, TL;DR means "too long; didn't read." These final sections will show you how to write tests for the features covered in each chapter and give a high-level overview of what was covered.

The book is written for Laravel 5.3, but because Laravel 5.1 is the latest LTS release, any features that are new in 5.2 or 5.3 will be identified.

# Conventions Used in This Book

The following typographical conventions are used in this book:

*Italic*

Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

**`Constant width bold`**

Shows commands or other text that should be typed literally by the user.

*`Constant width italic`*

Shows text that should be replaced with user-supplied values or by values determined by context.

> **TIP**
>
> This element signifies a tip or suggestion.

> **NOTE**
>
> This element signifies a general note.

> **WARNING**
>
> This element indicates a warning or caution.

# O’Reilly Safari

**NOTE**

*Safari* (formerly Safari Books Online) is membership-based training and reference platform for enterprise, government, educators, and individuals.

Members have access to thousands of books, training videos, Learning Paths, interactive tutorials, and curated playlists from over 250 publishers, including O’Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, and Course Technology, among others.

For more information, please visit *http://oreilly.com/safari*.

# How to Contact Us

Please address comments and questions concerning this book to the publisher:

- O’Reilly Media, Inc.
- 1005 Gravenstein Highway North
- Sebastopol, CA 95472
- 800-998-9938 (in the United States or Canada)
- 707-829-0515 (international or local)
- 707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/laravel-up-and-running*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

# Acknowledgments

This book would not have happened without the gracious support of my amazing wife Tereva or the understanding ("Daddy's writing, buddy!") of my son Malachi. And while she wasn't explicitly aware of it, my daughter Mia was around for almost the entire creation of the book, so this book is dedicated to the whole family. There were many, many long evening hours and weekend Starbucks trips that took me away from my family, and I couldn't be more grateful for their support and also their presence just making my life awesome.

Additionally, the entire Tighten Co. family has supported and encouraged me through the writing of the book, several even editing (Keith Damiani, editor extraordinaire) and helping me with challenging code samples (Adam Wathan, King of the Collection Pipeline). Dan Sheetz, my partner in Tighten crime, has been gracious enough to watch me while away many a work hour cranking on this book and was nothing but supportive and encouraging; and Dave Hicking, our operations manager, helped me arrange my schedule and work responsibilities around writing time.

Taylor Otwell deserves thanks and honor for creating Laravel — and therefore creating so many jobs and helping so many developers love our lives that much more. He deserves appreciation for how he's focused on developer happiness and how hard he's worked to have empathy for developers and to build a positive and encouraging community. But I also want to thank him for being a kind, encouraging, and challenging friend. Taylor, you're a boss.

Thanks to Jeffrey Way, who I still contend to be one of the best teachers on the Internet. He originally introduced me to Laravel and still introduces more people every day. He's also, unsurprisingly, a fantastic human being whom I'm glad to call a friend.

Thank you to Jess D'Amico, Shawn McCool, Ian Landsman, and Taylor for seeing value in me as a conference speaker early on and giving me a platform to teach from. Thanks to Dayle Rees for making it so easy for so many to learn Laravel in the early days.

Thanks to every person who put their time and effort into writing blog posts about Laravel, especially early on: Eric Barnes, Chris Fidao, Matt Machuga, Jason Lewis, Ryan Tablada, Dries Vints, Maks Surguy, and so many more.

And thanks to the entire community of friends on Twitter, IRC, and Slack who've interacted with me over the years. I wish I could name every name, but I would miss some and then feel awful about missing them. You all are brilliant, and I'm honored to get to interact with you on a regular basis.

Thanks to my O'Reilly editor, Ally MacDonald, and all of my technical editors: Keith Damiani, Michael Dyrynda, Adam Fairholm, and Myles Hyson.

And, of course, thanks to the rest of my family and friends, who supported me directly or indirectly through this process — my parents and siblings, the Gainesville community, other business owners and authors, other conference speakers, and the inimitable DCB. I need to stop writing because by the time I run out of space here I'll be thanking my Starbucks baristas.

# Chapter 1. Why Laravel?

In the early days of the dynamic web, writing a web application looked a lot different than it does today. Developers then were responsible for writing the code for not just the unique business logic of our applications, but also each of the components that are so common across sites — user authentication, input validation, database access, templating, and more.

Today, programmers have dozens of application development frameworks and thousands of components and libraries easily accessible. It's a common refrain among programmers that, by the time you learn one framework, three newer (and purportedly better) frameworks have popped up intending to replace it.

"Just because it's there" might be a valid justification for climbing a mountain, but there are better reasons to choose to use a specific framework — or to use a framework at all. It's worth asking the question: why frameworks? More specifically, why Laravel?

# Why Use a Framework?

It's easy to see why it's beneficial to use the individual components, or packages, that are available to PHP developers. With packages, someone else is responsible for developing and maintaining an isolated piece of code that has a well-defined job, and in theory that person has a deeper understanding of this single component than you have time to have.

Frameworks like Laravel — and Symfony, Silex, Lumen, and Slim — prepackage a collection of third-party components together with custom framework "glue" like configuration files, service providers, prescribed directory structures, and application bootstraps. So, the benefit of using a framework in general is that someone has made decisions not just about individual components for you, but also about *how those components should fit together.*

## “I’ll Just Build It Myself”

Let’s say you start a new web app without the benefit of a framework. Where do you begin? Well, it should probably route HTTP requests, so you now need to evaluate all of the HTTP request and response libraries available and pick one. Then a router. Oh, and you’ll probably need to set up some form of routes configuration file. What syntax should it use? Where should it go? What about controllers? Where do they live, and how are they loaded? Well, you probably need a dependency injection container to resolve the controllers and their dependencies. But which one?

Furthermore, what if you do take the time to answer all those questions and successfully create your application — what’s the impact on the next developer? What about when you have four such custom-framework–based applications, or a dozen, and you have to remember where the controllers live in each, or what the routing syntax is?

## Consistency and Flexibility

Frameworks address this issue by providing a carefully considered answer to the question “Which component should we use here?” and ensuring that the particular components chosen work well together. Additionally, frameworks provide conventions that reduce the amount of code a developer new to the project has to understand — if you understand how routing works in one Laravel project, for example, you understand how it works in all Laravel projects.

When someone prescribes rolling your own framework for each new project, what they’re really advocating is the ability to *control* what does and doesn’t go into your application’s foundation. That means the best frameworks will not only provide you with a solid foundation, but also give you the freedom to customize to your heart’s content. And this, as I’ll show you in the rest of this book, is part of what makes Laravel so special.

# A Short History of Web and PHP Frameworks

An important part of being able to answer the question “Why Laravel?” is understanding Laravel’s history — and understanding what came before it. Prior to Laravel’s rise in popularity, there were a variety of frameworks and other movements in PHP and other web development spaces.

## Ruby on Rails

David Heinemeier Hansson released the first version of Ruby on Rails in 2004, and it’s been hard to find a web application framework since then that hasn’t been influenced by Rails in some way.

Rails popularized MVC, RESTful JSON APIs, convention over configuration, ActiveRecord, and many more tools and conventions that had a profound influence on the way web developers approached their applications — especially with regard to rapid application development.

## The Influx of PHP Frameworks

It was clear to most developers that Rails, and similar web application frameworks, were the wave of the future, and PHP frameworks, including those admittedly imitating Rails, starting popping up quickly.

CakePHP was the first in 2005, and it was soon followed by Symfony, CodeIgniter, Zend Framework, and Kohana (a CodeIgniter fork). Yii arrived in 2008, and Aura and Slim in 2010. 2011 brought FuelPHP and Laravel, both of which were not quite CodeIgniter offshoots, but instead proposed as alternatives.

Some of these frameworks were more Rails-y, focusing on database object-relational mappers (ORMs), MVC structures, and other tools targeting rapid development. Others, like Symfony and Zend, focused more on enterprise design patterns and ecommerce.

## The Good and the Bad of CodeIgniter

CakePHP and CodeIgniter were the two early PHP frameworks that were most open about how much their inspiration was drawn from Rails. CodeIgniter quickly rose to fame and by 2010 was arguably the most popular of the independent PHP frameworks.

CodeIgniter was simple, easy to use, and boasted amazing documentation and a strong community. But its use of modern technology and patterns advanced slowly, and as the framework world grew and PHP's tooling advanced, CodeIgniter started falling behind in terms of both technological advances and out-of-the-box features. Unlike many other frameworks, CodeIgniter was managed by a company, and they were slow to catch up with PHP 5.3's newer features like namespaces and the moves to GitHub and later Composer. It was in 2010 that Taylor Otwell, Laravel's creator, became dissatisfied enough with CodeIgniter that he set off to write his own framework.

## Laravel 1, 2, and 3

The first beta of Laravel 1 was released in June 2011, and it was written completely from scratch. It featured a custom ORM (Eloquent); closure-based routing (inspired by Ruby Sinatra); a module system for extension; and helpers for forms, validation, authentication, and more.

Early Laravel development moved quickly, and Laravel 2 and 3 were released in November 2011 and February 2012, respectively. They introduced controllers, unit testing, a command-line tool, an inversion of control (IoC) container, Eloquent relationships, and migrations.

## Laravel 4

With Laravel 4, Taylor rewrote the entire framework from the ground up. By this point Composer, PHP's now-ubiquitous package manager, was showing signs of becoming an industry standard and Taylor saw the value of rewriting the framework as a collection of components, distributed and bundled together by Composer.

Taylor developed a set of components under the code name *Illuminate* and, in May 2013, released Laravel 4 with an entirely new structure. Instead of bundling the majority of its code as a download, Laravel now pulled in the majority of its components from Symfony (another framework that released its components for use by others) and the Illuminate components through Composer.

Laravel 4 also introduced queues, a mail component, facades, and database seeding. And because Laravel was now relying on Symfony components, it was announced that Laravel would be mirroring (not exactly, but soon after) the six-monthly release schedule Symfony follows.

## Laravel 5

Laravel 4.3 was scheduled to release in November 2014, but as development progressed, it became clear that the significance of its changes merited a major release, and Laravel 5 was released in February 2015.

Laravel 5 featured a revamped directory structure, removal of the form and HTML helpers, the introduction of the contract interfaces, a spate of new views, Socialite for social media authentication, Elixir for asset compilation, Scheduler to simplify cron, dotenv for simplified environment management, form requests, and a brand new REPL (read–evaluate–print loop).

# What’s So Special About Laravel?

So what is it that sets Laravel apart? Why is it worth having more than one PHP framework at any time? They all use components from Symfony anyway, right? Let’s talk a bit about what makes Laravel “tick.”

## The Philosophy of Laravel

You only need to read through the Laravel marketing materials and READMEs to start seeing its values. Taylor uses light-related words like “Illuminate” and “Spark.” And then there are these: “Artisans.” “Elegant.” Also, these: “Breath of fresh air.” “Fresh start.” And finally: “Rapid.” “Warp speed.”

The two most strongly communicated values of the framework are to increase developer speed and developer happiness. Taylor has described the “Artisan” language as intentionally contrasting against more utilitarian values. You can see the genesis of this sort of thinking in his 2011 question on StackExchange in which he stated, “Sometimes I spend ridiculous amounts of time (hours) agonizing over making code look pretty” — just for the sake of a better experience of looking at the code itself. And he’s often talked about the value of making it easier and quicker for developers to take their ideas to fruition, getting rid of unnecessary barriers to creating great products.

Laravel is, at its core, about equipping and enabling developers. Its goal is to provide clear, simple, and beautiful code and features that help developers quickly learn, start, and develop, and write code that’s simple, clear, and will last.

The concept of targeting developers is clear across Laravel materials. “Happy developers make the best code” is written in the documentation. “Developer happiness from download to deploy” was the unofficial slogan for a while. Of course, any tool or framework will say it wants developers to be happy. But having developer happiness as a *primary* concern, rather than secondary, has had a huge impact on Laravel’s style and decision-making progress. Where other frameworks may target architectural purity as their primary goal, or compatibility with the goals and values of enterprise development teams, Laravel’s primary focus is on serving the individual developer.

## How Laravel Achieves Developer Happiness

Just saying you want to make developers happy is one thing. Doing it is another, and it requires you to question what in a framework is most likely to make developers unhappy and what is most likely to make them happy. There are a few ways Laravel tries to make developers' lives easier.

First, Laravel is a rapid application development framework. That means it focuses on a shallow (easy) learning curve and on minimizing the steps between starting a new app and publishing it. All of the most common tasks in building web applications, from database interactions to authentication to queues to email to caching, are made simpler by the components Laravel provides. But Laravel's components aren't just great on their own; they provide a consistent API and predictable structures across the entire framework. That means that, when you're trying something new in Laravel, you're more than likely going to end up saying, "… and it just works."

This doesn't end at the framework itself, either. Laravel provides an entire ecosystem of tools for building and launching applications. You have Homestead and Valet for local development, Forge for server management, and Envoyer for advanced deployment. And there's a suite of add-on packages: Cashier for payments and subscriptions, Echo for WebSockets, Scout for search, Passport for API authentication, Socialite for social login, and Spark to bootstrap your SaaS. Laravel is trying to take the repetitive work out of developers' jobs so they can do something unique.

Next, Laravel focuses on "convention over configuration" — meaning that if you're willing to use Laravel's defaults, you'll have to do much less work than with other frameworks that require you to declare all of your settings even if you're using the recommended configuration. Projects built on Laravel take less time than those built on most other PHP frameworks.

Laravel also focuses deeply on simplicity. It's possible to use dependency injection and mocking and the Data Mapper pattern and repositories and Command Query Responsibility Segregation and all sorts of other more complex architectural patterns with Laravel, if you want. But while other frameworks might suggest using those tools and structures on every project, Laravel and its documentation and community lean toward starting with the simplest possible implementation — a global function here, a facade there, ActiveRecord over there. This allows developers to create the simplest possible application to solve for their needs.

An interesting source of how Laravel is different is that its creator and its community are more connected to and inspired by Ruby and Rails and functional programming languages than by Java. There's a strong current in modern PHP to lean toward verbosity and complexity, embracing the more Java-esque aspects of PHP. But Laravel tends to be on the other side, embracing expressive, dynamic, and simple coding practices and language features.

## The Laravel Community

If this book is your first exposure to the Laravel community, you have something special to look forward to. One of the distinguishing elements of Laravel, which has contributed to its growth and success, is the welcoming, teaching community that surrounds it. From Jeffrey Way's Laracasts video tutorials to Laravel News to Slack and IRC channels, from Twitter friends to bloggers to the Laracon conferences, Laravel has a rich and vibrant community full of folks who've been around since day one and folks who are on their own day one. And this isn't an accident:

> From the very beginning of Laravel, I've had this idea that all people want to feel like they are part of something. It's a natural human instinct to want to belong and be accepted into a group of other like-minded people. So, by injecting personality into a web framework and being really active with the community, that type of feeling can grow in the community.
> Taylor Otwell, *Product and Support interview*

Taylor understood from the early days of Laravel that a successful open source project needed two things: good documentation and a welcoming community. And those two things are now hallmarks of Laravel.

# How It Works

Up until now, everything I've shared here has been entirely abstract. What about the code, you ask? Let's dig into a simple application (Example 1-1) so you can see what working with Laravel day-to-day is actually like.

*Example 1-1. "Hello, World" in routes/web.php*

```
// File: routes/web.php
<?php

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

The simplest possible action you can take in a Laravel application is to define a route and return a result any time someone visits that route. If you initialize a brand new Laravel application on your machine, define the route in Example 1-1, and then serve the site from the *public* directory, you'll have a fully functioning "Hello, World" example (see Figure 1-1).

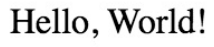



*Figure 1-1. Returning "Hello, World!" with Laravel*

It looks very similar to do the same with controllers, as you can see in Example 1-2.

*Example 1-2. "Hello, World" with controllers*

```
// File: routes/web.php
<?php

Route::get('/', 'WelcomeController@index');
// File: app/Http/Controllers/WelcomeController.php
<?php
namespace app\Http\Controllers;

class WelcomeController
{
    public function index()
    {
        return 'Hello, World!';
    }
}
```

And if we're storing our greetings in the database, it'll also look pretty similar (see Example 1-3).

*Example 1-3. Multigreeting "Hello, World" with database access*

```
// File: routes/web.php
<?php

Route::get('/', function() {
    return Greeting::first()->body;
});
```

```
// File: app/Greeting.php
<?php

use Illuminate\Database\Eloquent\Model;

class Greeting extends Model {}
// File: database/migrations/2015_07_19_010000_create_greetings_table.php
<?php

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

class CreateGreetingsTable extends Migration
{
    public function up()
    {
        Schema::create('greetings', function (Blueprint $table) {
            $table->increments('id');
            $table->string('body');
            $table->timestamps();
        });
    }

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

Example 1-3 might be a bit overwhelming, and if so, just skip over it. We'll learn about everything that's happening here in later chapters, but you can already see that with just a few lines of code, we've set up database migrations and models and pulled records out. It's just that simple.

# Why Laravel?

So — why Laravel?

Because Laravel helps you bring your ideas to reality with no wasted code, using modern coding standards, surrounded by a vibrant community, with an empowering ecosystem of tools.

And because you, dear developer, deserve to be happy.

# Chapter 2. Setting Up a Laravel Development Environment

Part of PHP's success has been because it's hard to find a web server that *can't* serve PHP. However, modern PHP tools have stricter requirements than those of the past. The best way to develop for Laravel is to ensure a consistent local and remote server environment for your code, and thankfully, the Laravel ecosystem has a few tools for this.

# System Requirements

Everything we'll cover in this chapter is possible with Windows machines, but you'll need dozens of pages of custom instructions and caveats. I'll leave those instructions and caveats to actual Windows users, so the examples here and in the rest of the book will focus on Unix/Linux/Mac OS developers.

Whether you choose to serve your website by installing PHP and other tools on your local machine, serve your development environment from a virtual machine via Vagrant, or rely on a tool like MAMP/WAMP/XAMPP, your development environment will need to have all of the following installed in order to serve Laravel sites:

- PHP >= 5.6.4 for Laravel 5.3 or PHP >= 5.5.9 for 5.1 and 5.2
- OpenSSL PHP extension
- PDO PHP extension
- Mbstring PHP extension
- Tokenizer PHP extension

# Composer

Whatever machine you're developing on will need to have Composer installed globally. If you're not familiar with Composer, it's a tool that's at the foundation of most modern PHP development. Composer is a dependency manager for PHP, much like NPM for Node or RubyGems for Ruby. You'll need Composer to install Laravel, update Laravel, and bring in external dependencies.

# Local Development Environments

For many projects, hosting your development environment using a simpler tool set will be enough. If you already have MAMP or WAMP or XAMPP installed on your system, that will likely be fine to run Laravel. You can also just run Laravel with PHP's built-in web server, assuming your system PHP is the right version.

All you really need *to get started* is the ability to run PHP. Everything past that is up to you.

However, Laravel offers two tools for local development, Valet and Homestead, and we'll cover both briefly. If you're unsure of which to use, I'd recommend using Valet and just skimming the Homestead section; however, both tools are valuable and worth understanding.

## Laravel Valet

If you want to use PHP's built-in web server, your simplest option is to serve every site from a *localhost* URL. If you run `php -S localhost:8000 -t public` from your Laravel site's root folder, PHP's built-in web server will serve your site at *http://localhost:8000/*. You can also run `php artisan serve` once you have your application set up to easily spin up an equivalent server.

But if you're interested in tying each of your sites to a specific development domain, you'll need to get comfortable with your operating system's hosts file and use a tool like dnsmasq. Let's instead try something simpler.

If you're a Mac user (there are also unofficial forks for Windows and Linux), Laravel Valet takes away the need to connect your domains to your application folders. Valet installs dnsmasq and a series of PHP scripts that make it possible to type `laravel new myapp && open myapp.dev` and for it to *just work*. You'll need to install a few tools using Homebrew, which the documentation will walk you through, but the steps from initial installation to serving your apps are few and simple.

Install Valet (see the docs for the latest installation instruction — it's under very active development at this time of writing), and point it at one or more directories where your sites will live. I ran `valet park` from my *~/Sites* directory, which is where I put all of my under-development apps. Now, you can just add *.dev* to the end of the directory name and visit it in your browser.

Valet makes it easy to serve all folders in a given folder as "FOLDERNAME.dev" using `valet park`, to serve just a single folder using `valet link`, to open the Valet-served domain for a folder using `valet open`, to serve the Valet site with HTTPS using `valet secure`, and to open an ngrok tunnel so you can share your site with others with `valet share`.

## Laravel Homestead

Homestead is another tool you might want to use to set up your local development environment. It's a configuration tool that sits on top of Vagrant and provides a pre-configured virtual machine image that is perfectly set up for Laravel development, *and* mirrors the most common production environment that many Laravel sites run on.

### Setting up Homestead

If you choose to use Homestead, it's going to take a bit more work to set up than something like MAMP or Valet. The benefits are myriad, however: configured correctly, your local environment can be incredibly close to your remote working environment; you won't have to worry about updating your dependencies on your local machine; and you can learn all about the structure of Ubuntu servers from the safety of your local machine.

**WHAT TOOLS DO HOMESTEAD OFFER?**

You can always upgrade the individual components of your Homestead virtual machine, but here are a few important tools Homestead comes with by default:

- To run the server and serve the site, Ubuntu, PHP, and Nginx (a web server similar to Apache).
- For database/storage and queues, MySQL, Postgres, Redis, Memcached, and *beanstalkd*.
- For build steps and other tools, Node.

### Installing Homestead's dependencies

First, you'll need to download and install either VirtualBox or VMWare. VirtualBox is most common because it's free.

Next, download and install Vagrant.

Vagrant is convenient because it makes it easy for you to create a new local virtual machine from a precreated "box," which is essentially a template for a virtual machine. So, the next step is to run `vagrant box add laravel/homestead` from your terminal to download the box.

### Installing Homestead

Next, let's actually install Homestead. You can install multiple instances of Homestead (perhaps hosting a different Homestead box per project), but I prefer a single Homestead virtual machine for all of my projects. If you want one per project, you'll want to install Homestead in your project directory; check the Homestead documentation online for instructions. If you want a single virtual machine for all of your projects, install Homestead in your user's home directory using the following command:

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

Now, run the initialization script from wherever you put the *Homestead* directory:

```
bash ~/Homestead/init.sh
```

This will place Homestead's primary configuration file, *Homestead.yaml*, in a new *~/.homestead* directory.

### Configuring Homestead

Open up *Homestead.yaml* and configure it how you'd like. Here's what it looks like out of the box:

```
ip: "192.168.10.10"
memory: 2048
cpus: 1
provider: virtualbox

authorize: ~/.ssh/id_rsa.pub

keys:
    - ~/.ssh/id_rsa

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

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

databases:
    - homestead

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

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

You'll need to tell it your provider (likely `virtualbox`), point it to your public SSH key (by default `~/.ssh/id_rsa.pub`; if you don't have one, GitHub has a great tutorial on creating SSH keys), map folders and sites to their local machine equivalents, and provision a database.

Mapping folders in Homestead allows you to edit files on your local machine and have those files show up in your Vagrant box so they can be served. For example, if you have a *~/Sites* directory where you put all of your code, you'd map the folders in Homestead by replacing the folders section in *Homestead.yaml* with the following:

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

You've now just created a directory in your Homestead virtual machine at */home/vagrant/Sites* that will mirror your computer's directory at *~/Sites*.

### TOP-LEVEL DOMAINS FOR DEVELOPMENT SITES

You can choose any convention for local development sites' URLs, but *.app* and *.dev* are the most common. Homestead suggests `.app`, so if I'm working on a local copy of *symposiumapp.com*, I'll develop at *symposiumapp.app*.

Now, let's set up our first example website. Let's say our live site is going to be *projectName.com*. In *Homestead.yaml*, we'll map our local development folder to *projectName.app*, so we have a separate URL to visit for local development:

```
sites:
    - map: projectName.app
      to: /home/vagrant/Sites/projectName/public
```

As you can see, we're mapping the URL *projectName.app* to the virtual machine directory */home/vagrant/Sites/projectName/public*, which is the *public* folder within our Laravel install. We'll learn more about that later.

Finally, you're going to need to teach your local machine that, when you try to visit *projectName.app*, it should look at your computer's local IP address to resolve it. Mac and Linux users should edit */etc/hosts*, and Windows users *C:\Windows\System32\drivers\etc\hosts*. Add a line to this file that looks like this:

```
192.168.10.10  projectName.app
```

Once you've provisioned Homestead, your site will be available to browse (on your machine) at *http://projectName.app/*.

### Creating databases in Homestead

Just like you can define a site in *Homestead.yaml*, you can also define a database. Databases are a lot simpler, because you're only telling the provisioner to *create* a database with that name, nothing else. We do this as follows:

```
databases:
    - projectname
```

### Provisioning Homestead

The first time you actually turn on a Homestead box, you need to tell Vagrant to initialize it. Navigate to your *Homestead* directory and run `vagrant up`:

```
cd ~/Homestead
vagrant up
```

Your Homestead box is now up and running; it's mirroring a local folder, and it's serving it to a URL you can visit in any browser on your computer. It also has created a MySQL database. Now that you have that environment running, you're ready to set up your first Laravel project — but first, a quick note about using Homestead day-to-day.

### Using Homestead day-to-day

It's common to leave your Homestead virtual machine up and running at all times, but if you don't, or if you have recently restarted your computer, you'll need to know how to spin the box up and down.

Since Homestead is based on Vagrant commands, you'll just use basic Vagrant commands for most Homestead actions. Change to the directory where you installed Homestead (using `cd`) and then run the following commands as needed:

- `vagrant up` spins up the Homestead box.
- `vagrant suspend` takes a snapshot of where the box is and then shuts it down; like "hibernating" a desktop machine.
- `vagrant halt` shuts the entire box down; like turning off a desktop machine.
- `vagrant destroy` deletes the entire box; like formatting a desktop machine.
- `vagrant provision` re-runs the provisioners on the preexisting box.

### Connecting to Homestead databases from desktop applications

If you use a desktop application like Sequel Pro, you'll likely want to connect to your Homestead MySQL databases from your host machine. These settings will get you going:

- **Connection type:** Standard (non-SSH)
- **Host:** 127.0.0.1
- **Username:** homestead
- **Password:** secret
- **Port:** 33060

# Creating a New Laravel Project

There are two ways to create a new Laravel project, but both are run from the command line. The first option is to globally install the Laravel installer tool (using Composer); the second is to use Composer's `create-project` feature.

You can learn about both options in more detail on the Installation documentation page, but I'd recommend the Laravel installer tool.

## Installing Laravel with the Laravel Installer Tool

If you have Composer installed globally, installing the Laravel installer tool is as simple as running the following command:

```
composer global require "laravel/installer=~1.1"
```

Once you have the Laravel installer tool installed, spinning up a new Laravel project is simple. Just run this command from your command line:

```
laravel new projectName
```

This will create a new subdirectory of your current directory named *projectName* and install a bare Laravel project in it.

## Installing Laravel with Composer's create-project Feature

Composer also offers a feature called `create-project` for creating new projects with a particular skeleton. To use this tool to create a new Laravel project, issue the following command:

```
composer create-project laravel/laravel projectName --prefer-dist
```

Just like the installer tool, this will create a subdirectory of your current directory named *projectName* that contains a skeleton Laravel install, ready for you to develop.

# Laravel’s Directory Structure

When you open up a directory that contains a skeleton Laravel application, you’ll see the following files and directories:

```
app/
bootstrap/
config/
database/
public/
resources/
routes/
storage/
tests/
vendor/
.env
.env.example
.gitattributes
.gitignore
artisan
composer.json
composer.lock
gulpfile.js
package.json
phpunit.xml
readme.md
server.php
```

Let’s walk through them one by one to get familiar.

## The Folders

The root directory contains the following folders by default:

- *app* is where the bulk of your actual application will go. Models, controllers, route definitions, commands, and your PHP domain code all go in here.
- *bootstrap* contains the files that the Laravel framework uses to boot every time it runs.
- *config* is where all the configuration files live.
- *database* is where database migrations and seeds live.
- *public* is the directory the server points to when it's serving the website. This contains *index.php*, which is the front controller that kicks off the bootstrapping process and routes all requests appropriately. It's also where any public-facing files like images, stylesheets, scripts, or downloads go.
- *resources* is where non-PHP files that are needed for other scripts live. Views, language files, and (optionally) Sass/LESS and source JavaScript files live here.
- *routes* is where all of the route definitions live, both for HTTP routes and "console routes," or Artisan commands.
- *storage* is where caches, logs, and compiled system files live.
- *tests* is where unit and integration tests live.
- *vendor* is where Composer installs its dependencies. It's Git-ignored (marked to be excluded from your version control system), as Composer is expected to run as a part of your deploy process on any remote servers.

## The Loose Files

The root directory also contains the following files:

- *.env* and *.env.example* are the files that dictate the environment variables (variables that are expected to be different in each environment and are therefore not committed to version control). *.env.example* is a template that each environment should duplicate to create its own *.env* file, which is Git-ignored.
- *artisan* is the file that allows you to run Artisan commands (see Chapter 7) from the command line.
- *.gitignore* and *.gitattributes* are Git configuration files.
- *composer.json* and *composer.lock* are the configuration files for Composer; *composer.json* is user-editable and *composer.lock* is not. These files share some basic information about this project and also define its PHP dependencies.
- *gulpfile.js* is the (optional) configuration file for Elixir and Gulp. This is for compiling and processing your frontend assets.
- *package.json* is like *composer.json* but for frontend assets.
- *phpunit.xml* is a configuration file for PHPUnit, the tool Laravel uses for testing out of the box.
- *readme.md* is a Markdown file giving a basic introduction to Laravel.
- *server.php* is a backup server that tries to allow less-capable servers to still preview the Laravel application.

# Configuration

The core settings of your Laravel application — database connection, queue and mail settings, etc. — live in files in the *config* folder. Each of these files returns an array, and each value in the array will be accessible by a config key that is comprised of the filename and all descendant keys, separated by dots (.)

So, if you create a file at *config/services.php* that looks like this:

```
// config/services.php
return [
    'sparkpost' => [
        'secret' => 'abcdefg'
    ]
];
```

you will now have access to that config variable using `config('services.sparkpost.secret')`.

Any configuration variables that should be distinct for each environment (and therefore not committed to source control) will instead live in your *.env* files. Let's say you want to use a different Bugsnag API key for each environment. You'd set the config file to pull it from *.env*:

```
// config/services.php
return [
    'bugsnag' => [
        'api_key' => env('BUGSNAG_API_KEY')
    ]
];
```

This `env()` helper function pulls a value from your *.env* file with that same key. So now, add that key to your *.env* (settings for this environment) and *.env.example* (template for all environments) files:

```
BUGSNAG_API_KEY=oinfp9813410942
```

Your *.env* file already contains quite a few environment-specific variables needed by the framework, like which mail driver you'll be using and what your basic database settings are.

# Up and Running

You're now up and running with a bare Laravel install. Run `git init`, commit the bare files with `git add .` and `git commit`, and you're ready to start coding. That's it! And if you're using Valet, you can run the following commands and instantly see your site live in your browser:

```
laravel new myProject && cd myProject && valet open
```

Every time I start a new project, these are the steps I take:

```
laravel new myProject
cd myProject
git init
git add .
git commit -m "Initial commit"
```

I keep all of my sites in a *~/Sites* folder, which I have set up as my primary Valet directory, so in this case I'd instantly have *myProject.dev* accessible in my browser with no added work. I can edit *.env* and point it to a particular database, add that database in my MySQL app, and I'm ready to start coding.

## LAMBO

I perform the this set of steps so often that I created a simple global Composer package to do it for me. It’s called Lambo, and you can learn more about it on GitHub.

# Testing

In every chapter after this, the "Testing" section at the end of the chapter will show you how to write tests for the feature or features that were covered. Since this chapter doesn't cover a testable feature, let's talk tests quickly. (To learn more about writing and running tests in Laravel, head over to Chapter 12.)

Out of the box, Laravel brings in PHPUnit as a dependency and is configured to run the tests in any file in the *tests* directory whose name ends with *Test.php* (for example, *tests/UserTest.php*).

So, the simplest way to write tests is to create a file in the *tests* directory with a name that ends with *Test.php*. And the easiest way to run them is to run `./vendor/bin/phpunit` from the command line (in the project root).

If any tests require database access, be sure to run your tests from the machine where your database is hosted — so if you're hosting your database in Vagrant, make sure to `ssh` into your Vagrant box to run your tests from there. Again, you can learn about this and much more in Chapter 12.

# TL;DR

Since Laravel is a PHP framework, it’s very simple to serve it locally. Laravel also provides two tools for managing your local development: a simpler tool called Valet that uses your local machine to provide your dependencies, and a preconfigured Vagrant setup named Homestead. Laravel relies on, and can be installed by, Composer, and comes out of the box with a series of folders and files that reflect both its conventions and its relationship with other open source tools.

# Chapter 3. Routing and Controllers

The essential function of any web application framework is to take requests from a user and deliver responses, usually via HTTP(S). This means defining an application's routes is the first and most important project to tackle when learning a web framework; without routes, you have no ability to interact with the end user.

In this chapter we will examine routes in Laravel and show how to define them, how to point them to the code they should execute, and how to use Laravel's routing tools to handle a diverse array of routing needs.

# Route Definitions

In a Laravel application, you will define your "web" routes in *routes/web.php* and your "API" routes in *routes/api.php*. Web routes are those that will be visited by your end users; API routes are those for your API, if you have one. For now, we'll primarily focus on the routes in *routes/web.php*.

> **NOTE**
>
> In projects running versions of Laravel prior to 5.3, there will be only one routes file, located at *app/Http/routes.php*.

The simplest way to define a route is to match a path (e.g., `/`) with a closure, as seen in Example 3-1.

*Example 3-1. Basic route definition*

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

> **WHAT'S A CLOSURE?**
>
> Closures are PHP's version of anonymous functions. A closure is a function that you can pass around as an object, assign to a variable, pass as a parameter to other functions and methods, or even serialize.

You've now defined that, if anyone visits `/` (the root of your domain), Laravel's router should run the closure defined there and return the result. Note that we `return` our content and don't `echo` or `print` it.

### A QUICK INTRODUCTION TO MIDDLEWARE

You might be wondering, “Why am I returning ‘Hello, World!’ instead of echoing it?”

There are quite a few answers, but the simplest is that there are a lot of wrappers around Laravel’s request and response cycle, including something called middleware. When your route closure or controller method is done, it’s not time to send the output to the browser yet; returning the content allows it to continue flowing through the response stack and the middleware before it is returned back to the user.

Many simple websites could be defined entirely within the web routes file. With a few simple `GET` routes combined with some templates as illustrated in Example 3-2, you can can serve a classic website easily.

*Example 3-2. Sample website*

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

Route::get('about', function () {
    return view('about');
});

Route::get('products', function () {
    return view('products');
});

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

### STATIC CALLS

If you have much experience developing PHP, you might be surprised to see static calls on the `Route` class. This is not actually a static method per se, but rather service location using Laravel's facades, which we'll cover in Chapter 11.

If you prefer to avoid facades, you can accomplish these same definitions like this:

```
$router->get('/', function () {
    return 'Hello, World!';
});
```

### HTTP METHODS

If you're not familiar with the idea of HTTP methods, read on in this chapter for more information, but for now, just know that every HTTP request has a "verb," or action, along with it. Laravel allows you to define your routes based on which verb was used; the most common are `GET` and `POST`, followed by `PUT`, `DELETE`, and `PATCH`. Each method communicates a different thing to the server, and to your code, about the intentions of the caller.

## Route Verbs

You might've noticed that we've been using `Route::get` in our route definitions. This means we're telling Laravel to only match for these routes when the HTTP request uses the `GET` action. But what if it's a form `POST`, or maybe some JavaScript sending `PUT` or `DELETE` requests? There are a few other options for methods to call on a route definition, as illustrated in Example 3-3.

*Example 3-3. Route verbs*

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

Route::post('/', function () {});

Route::put('/', function () {});

Route::delete('/', function () {});

Route::any('/', function () {});

Route::match(['get', 'post'], '/', function () {});
```

## Route Handling

As you've probably guessed, passing a closure to the route definition is not the only way to teach it how to resolve a route. Closures are quick and simple, but the larger your application gets, the clumsier it becomes to put all of your routing logic in one file. Additionally, applications using route closures can't take advantage of Laravel's route caching (more on that later), which can shave up to hundreds of milliseconds off of each request.

The other common option is to pass a controller name and method as a string in place of the closure, as in Example 3-4.

*Example 3-4. Routes calling controller methods*

```
Route::get('/', 'WelcomeController@index');
```

This is telling Laravel to pass requests to that path to the `index()` method of the `App\Http\Controllers\WelcomeController` controller. This method will be passed the same parameters and treated the same way as a closure you might've alternatively put in its place.

## Route Parameters

If the route you're defining has parameters — segments in the URL structure that are variable — it's simple to define them in your route and pass them to your closure (see Example 3-5).

*Example 3-5. Route parameters*

```
Route::get('users/{id}/friends', function ($id) {
    //
});
```

> **THE NAMING RELATIONSHIP BETWEEN ROUTE PARAMETERS AND CLOSURE/CONTROLLER METHOD PARAMETERS**
>
> As you can see in Example 3-5, it's most common to use the same names for your route parameters (`{id}`) and the method parameters they inject into your route definition (`function ($id)`). But is this necessary?
>
> Unless you're using route/model binding, no. The only thing that defines which route parameter matches with which method parameter is their order (left to right), as you can see here:
>
> ```
> Route::get('users/{userId}/comments/{commentId}', function (
>     $thisIsActuallyTheRouteId,
>     $thisisReallyTheCommentId
>     ) {
>     //
> });
> ```
>
> That having been said, just because you *can* make them different doesn't mean you *should*. I recommend keeping them the same for the sake of future developers, who could get tripped up by inconsistent naming.

You can also make your route parameters optional by including a question mark (`?`) after the parameter name, as illustrated in Example 3-6. In this case, you should also provide a default value for the route's corresponding variable.

*Example 3-6. Optional route parameters*

```
Route::get('users/{id?}', function ($id = 'fallbackId') {
    //
});
```

And you can use regular expressions (regexes) to define that a route should only match if a parameter meets particular requirements, as in Example 3-7.

*Example 3-7. Regular expression route constraints*

```
Route::get('users/{id}', function ($id) {
    //
})->where('id', '[0-9]+');

Route::get('users/{username}', function ($username) {
    //
})->where('username', '[A-Za-z]+');

Route::get('posts/{id}/{slug}', function ($id, $slug) {
    //
})->where(['id' => '[0-9]+', 'slug' => '[A-Za-z]+']);
```

As you've probably guessed, if you visit a path that matches a route string, but the regex doesn't match the parameter, it won't be matched. Since routes are matched top to bottom, `users/abc` would skip the first closure in Example 3-7, but it would be matched by the second closure, so it would get routed there. On the other hand, `posts/abc/123` wouldn't match any of the closures, so it would return a 404 Not Found error.

## Route Names

The simplest way to refer to these routes elsewhere in your application is just by their path. There's a `url()` helper to simplify that linking in your views, if you need it; see Example 3-8 for an example. The helper will prefix your route with the full domain of your site.

*Example 3-8. URL helper*

```
<a href="<?php echo url('/'); ?>">
// outputs <a href="http://myapp.com/">
```

However, Laravel also allows you to name each route, which enables you to refer to it without explicitly referencing the URL. This is helpful because it means you can give simple nicknames to complex routes, and also because linking them by name means you don't have to rewrite your frontend links if the paths change (see Example 3-9).

*Example 3-9. Defining route names*

```
// Defining a route with name in routes/web.php:
Route::get('members/{id}', 'MembersController@show')->name('members.show');

// Link the route in a view using the route() helper
<a href="<?php echo route('members.show', ['id' => 14]); ?>">
```

This example illustrates a few new concepts. First, we're using fluent route definition to add the name, by chaining the `name()` method after the `get()` method. This method allows us to name the route, giving it a short alias to make it easier to reference elsewhere.

### DEFINING CUSTOM ROUTES IN LARAVEL 5.1

Fluent route definitions don't exist in Laravel 5.1. You'll need to instead pass an array to the second parameter of your route definition; check the Laravel docs to see more about how this works. Here's Example 3-9 in Laravel 5.1:

```
Route::get('members/{id}', [
    'as' => 'members.show',
    'uses' => 'MembersController@show'
]);
```

In our example, we've named this route `members.show`; *`resourcePlural.action`* is a common convention within Laravel for route and view names.

### ROUTE NAMING CONVENTIONS

You can name your route anything you'd like, but the common convention is to use the plural of the resource name, then a period, then the action. So, here are the routes most common for a resource named `photo`:

```
photos.index
photos.create
photos.store
photos.show
photos.edit
photos.update
photos.destroy
```

To learn more about these conventions, see "Resource Controllers".

We also introduced the `route()` helper. Just like `url()`, it's intended to be used in views to simplify linking to a named route. If the route has no parameters, you can simply pass the route name: (`route('members.index')`) and receive a route string `http://myapp.com/members/index`). If it has parameters, pass them in as an array as the second parameter like we did in this example.

In general, I recommend using route names instead of paths to refer to your routes, and therefore using the `route()` helper instead of the `url()` helper. Sometimes it can get a bit clumsy — for example, if you're working with multiple subdomains — but it provides an incredible level of flexibility to later change the application's routing structure without major penalty.

## PASSING ROUTE PARAMETERS TO THE ROUTE() HELPER

When your route has parameters (e.g., `users/{id}`), you need to define those parameters when you're using the `route()` helper to generate a link to the route.

There are a few different ways to pass these parameters. Let's imagine a route defined as `users/{userId}/comments/{commentId}`. If the user ID is `1` and the comment ID is `2`, let's look at a few options we have available to us:

**Option 1:**

```
route('users.comments.show', [1, 2])
// http://myapp.com/users/1/comments/2
```

**Option 2:**

```
route('users.comments.show', ['userId' => 1, 'commentId' => 2])
// http://myapp.com/users/1/comments/2
```