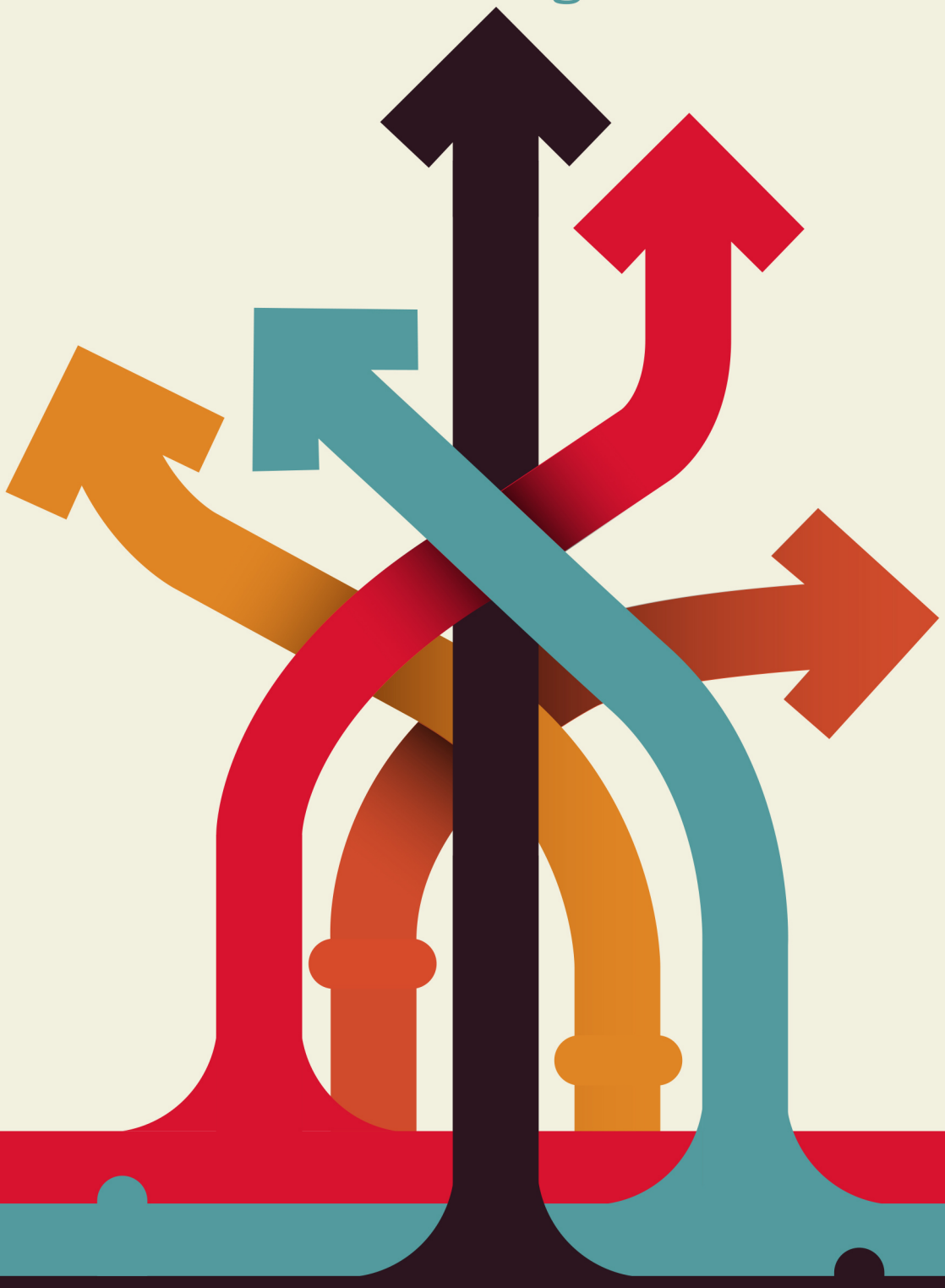
EASY LARAVEL 5

A Hands On Introduction Using a Real-World Project



W. JASON GILMORE

easylaravelbook.com

Easy Laravel 5

A Hands On Introduction Using a Real-World Project

W. Jason Gilmore

This book is for sale at http://leanpub.com/easylaravel

This version was published on 2018-02-16

Also By W. Jason Gilmore

Easy Active Record for Rails Developers

Easy E-Commerce Using Laravel and Stripe

Easy React

Dedicated to The Champ, The Princess, and Little Winnie. Love, Daddy

Contents

Introduction

I've spent the vast majority of my professional career (20 years and counting) immersed in the PHP language. During this time I've written eight PHP-related books, including a few bestsellers. Along the way I've worked on dozens of PHP-driven applications for clients ranging from unknown startups to globally-recognized companies, penned hundreds of articles about PHP and web development for some of the world's most popular print and online publications, and personally trained hundreds of developers on various PHP-related topics. In short, over the span of two decades I've pretty much seen it all when it comes to PHP.

So it might come as a surprise to some that I've never been more immersed in the language than right now. The PHP community and project ecosystem has never been stronger than it is today, offering an incredible number of libraries, frameworks, and tools which allow PHP developers to build more complicated web applications faster than they ever have before. And at least at the time of this writing there is no more popular PHP project on the planet than the *Laravel framework*.

Over the past several years I've worked on multiple large Laravel projects for a variety of clientele. Among others these projects include a REST API for an incredibly popular iOS and Android app, an e-commerce application for selling subscription-based services, and a huge intranet application for a South American agricultural concern. I can say without hesitation that these projects have ranked among the most entertaining and fulfilling in my career, and that sentiment has largely to do with the incredible power and productivity Laravel bestows upon its users.

This book summarizes all of the hard-won knowledge and experience I've amassed building these Laravel projects, and indeed that accrued building web applications of all shapes and sizes over the past two decades. By its conclusion, you'll have gained a well rounded understanding of Laravel's many features, and along the way will have been introduced to many best practices pertaining to code organization, testing, and deployment. I can't wait to get started!

Introducing the HackerPair Companion Project

Too many programming tutorials skew far more heavily in favor of academic exercises than real-world practicalities. Not so in this book. A significant amount of the material found herein is based upon the development of a project called HackerPair (http://hackerpair.com) which you can interact with *right now* by heading over to the HackerPair website (http://hackerpair.com).

HackerPair incorporates many, if not all, of the Laravel features you'll want to be acquainted with when building your own applications. I'll highlight just a few of the features here. Keep in mind however that not all of these features are currently available in the beta release, although they'll be added soon!

- **Comprehensive Database Seeding**: Many beginning developers tend to skip over the generation of real-world data for the development environment. HackerPair includes extensive data generation scripts (known as seeds) for users, events, categories, and locations.
- **Rigorous Form Integration and Validation**: HackerPair uses the powerful LaravelCollective/HTML package for forms generation, and relies on formalized Laravel procedures for input validation including use of the native Laravel validators and form requests.
- **Extensive Model Relationships**: HackerPair offers numerous examples of model relationships by including features such as event creation (events are owned by users), event favorites (users can favorite many events), event locations (events belong to states, states have many events), and so on.
- **User Authentication and Profile Management**: Laravel offers great off-the-shelf support for user registration and login, however developers will quickly outgrow the defaults. HackerPair extends the registration form to include several additional fields, extensively modifies the default registration and login view formatting, and adds account profile management.
- **Social Login**: In addition to standard user registration and authentication, users can instead opt to login using a third-party service such as GitHub and Twitter.
- **Vue.js Features**: Vue.js is Laravel's de facto JavaScript library. HackerPair includes a number of cool Vue.js features, including AJAX-driven event favoriting, event attendance management, and notifications.
- **Bootstrap 4 Integration**: Although Bootstrap 4 is still in beta at the time of this writing, I wanted to give it a spin and am glad I did. Although I will make clear I'm not exactly a CSS guru, and you'll probably find some of my styling to be repulsive. At any rate, in the book you'll also learn how to integrate Bootstrap 3 and the new Tailwind CSS frameworks.
- **Extensive Automated Testing**: One of my favorite Laravel features is the practically push button automated test integration. The HackerPair project includes extensive testing of numerous aspects of the code, including unit tests, model tests, and integration tests using Laravel Dusk.
- **A REST API**: We want to give developers the chance to build their own cool HackerPair applications, and so have exposed a REST API which allows information about events to be retrieved for display in a variety of formats.

Best of all, all interested Laravel developers are able to peruse and download the HackerPair GitHub repository for free! Head on over to http://github.com/wjgilmore/hackerpair to view the code.

About this Book

This book is broken into 12 chapters, each of which is briefly described below. Remember, the book is currently in beta, which is why not all of these chapters are included in your download! Many are under development and almost complete, so in the coming weeks I'll be regularly pushing up new versions and notifying readers.

Chapter 1. Introducing Laravel

In this opening chapter you'll learn how to create and configure your Laravel project using your existing PHP development environment, a virtual machine known as Laravel Homestead, and a minimal development environment known as Valet (OSX users only). I'll also show you how to configure your environment in order to effectively debug your Laravel applications, and how to extend Laravel's capabilities by installing several popular third-party packages. We'll conclude the chapter with an introduction to PHPUnit, showing you how to create and execute your first automated Laravel test!

Chapter 2. Managing Your Project Controllers, Layout, Views, and Other Assets

In this chapter you'll learn how to create controllers and actions, and define the routes used to access your application endpoints. You'll also learn how to create the pages (views), work with variable data and logic using the Blade templating engine, and reduce redundancy using layouts and view helpers. I'll also introduce Laravel Elixir, a new feature for automating otherwise laborious tasks such as JavaScript transpiling and CSS minification. You'll also learn how to integrate several popular CSS frameworks, including Bootstrap 3 and 4, and Tailwind, and how to use Laravel Dusk for integration testing.

Chapter 3. Talking to the Database

In this chapter we'll turn our attention to the project's data. You'll learn how to integrate and configure the database, manage your database schema using migrations, and easily populate your database using seeds. From there we'll move on to creating models, and how to query the database through these models using the Eloquent object relational mapper. I'll also introduce the concept of resourceful controllers, and we'll generate a controller which will be used to view and manage the example project's events. You'll also learn how how to use Laravel's Query Builder to query the database when Eloquent isn't possible or practical.

Chapter 4. Customizing Your Models

Laravel models are incredibly powerful tools, and can be customized in a variety of ways to meet your project's specific needs. In this chapter you'll learn how to override model defaults to create custom accessors and mutators, add instance methods, and use scopes to easily filter database results with minimal code redundancy. You'll also learn how to create sluggable URLs using the eloquent-sluggable package. The chapter concludes with an introduction to testing your models using Laravel's amazing database-specific test features.

Chapter 5. Creating, Updating, and Deleting Data

Chapters 3 and 4 were primarily focused upon the many different ways you can query the database. In this chapter we'll turn our attention to creating, updating, and deleting data. In addition to a review of the Eloquent syntax used to perform these tasks, we'll continue building out the resourceful controller created in chapter 3. You'll also learn how to incorporate flash notifications into your controllers and views to keep users updated regarding request outcomes, and how to use Laravel Dusk to test your forms.

Chapter 6. Validating User Input

For reasons of simplicity, chapter 5 focused exclusively on what it must be like to live in a world in which error prone or malicious users didn't exist. That is to say I momentarily punted on the matter of user input validation. But data validation is so crucial to successful web application development that it can be put off no longer, and so this chapter is devoted entirely to the topic. In this chapter you'll learn all about Laravel's native validators, and how to incorporate form requests into your project to add form validation while ensuring your controller code remains lean.

Chapter 7. Creating and Managing Model Relationships

Building and navigating table relations is a standard part of the development process even when working on the most unambitious of projects, yet this task is often painful when working with many web frameworks. Fortunately, using Laravel it's easy to define and traverse these relations. In this chapter I'll show you how to define, manage, and interact with one-to-one, one-to-many, many-to-many, has many through, and polymorphic relations.

Chapter 8. Sending E-mails

Whether for requiring newly registered users to confirm their e-mail address, or notifying event attendees of scheduling changes, web applications such as HackerPair rely heavily on using e-mail as an efficient means of communication. In this chapter you'll learn about Laravel's `Mailable` class, a fantastic solution for generating e-mails within your application. You'll also learn how to test e-mail generation and delivery in a sane fashion. Just for added measure, I'll walk you through the steps I took to incorporate a contact form into HackerPair, which when submitted, sends inquiring users' messages and contact details to a support address.

Chapter 9. Authenticating and Managing Your Users

Most modern applications offer user registration and preference management features in order to provide customized, persisted content and settings. In this chapter you'll learn how to integrate user registration, login, and account management capabilities into your Laravel application. I'll also show you how to add social authentication to your application, allowing users to authenticate using a variety of popular OAuth providers such as Twitter, Facebook, and GitHub.

Chapter 10. Creating an Administration Console

Most web applications incorporate a restricted administration console accessible by the project developers and support team. In this chapter I'll show you an easy solution for designating certain users as administrators, and how to grant access to a restricted console using prefixed route grouping and custom middleware.

Chapter 11. Introducing Vue.js

Vue.js[1] has become the Laravel community's de facto JavaScript library, and for good reason; it shares many of the practical, productive attributes Laravel developers have come to love. Chapter 13 introduces Vue.js' fundamental features, and shows you how to integrate highly interactive and eye-appealing interfaces into your Laravel application.

Chapter 12. Creating an Application API

These days a web interface is often only one of several available vehicles for interacting with the underlying data. Popular services such as GitHub, Amazon, and Google also offer an API (Application Programming Interface) which allows enterprising developers to dream up and implement new ways to view, mine, and update these companies' vast data stores. In this chapter you'll learn how to create your own API, and provide registered users with an API key which they'll use to authenticate when interacting with the API.

About W. Jason Gilmore

I'm W. Jason Gilmore[2], a software developer, consultant, and bestselling author. I've spent much of the past 17 years helping companies of all sizes build amazing technology solutions. Recent projects include an API for one of the world's highest volume robocall blockers, a SaaS for the interior design and architecture industries, an intranet application for a major South American avocado farm, an e-commerce analytics application for a globally recognized publisher, and a 10,000+ product online store for the environmental services industry.

I'm the author of eight books, including the bestselling *Beginning PHP and MySQL, Fourth Edition*, *Easy E-Commerce Using Laravel and Stripe* (with co-author and Laravel News founder Eric L. Barnes), and *Easy Active Record for Rails Developers*.

Over the years I've published more than 300 articles within popular publications such as Developer.com, JSMag, and Linux Magazine, and instructed hundreds of students in the United States and Europe. I'm also cofounder of the wildly popular CodeMash Conference[3], the largest multi-day developer event in the Midwest.

[1] http://vuejs.org/

[2] http://www.wjgilmore.com

[3] http://www.codemash.org

Away from the keyboard, you'll often find me playing with his kids, thinking about chess, and having fun with DIY electronics.

I love talking to readers and invite you to e-mail me at wj@wjgilmore.com.

Errata and Suggestions

Nobody is perfect, particularly when it comes to writing about technology. I've surely made some mistakes in both code and grammar, and probably completely botched more than a few examples and explanations. If you would like to report an error, ask a question or offer a suggestion, please e-mail me at wj@wjgilmore.com.

Chapter 1. Introducing Laravel

Laravel is a web application framework that borrows from the very best features of other popular framework solutions, among them Ruby on Rails and ASP.NET MVC. For this reason, if you have any experience working with other frameworks then I'd imagine you'll make a pretty graceful transition to Laravel. Newcomers to framework-driven development will have a slightly steeper learning curve due to the introduction of new concepts. I promise Laravel's practical and user-friendly features will make your journey an enjoyable one.

In this chapter you'll learn how to install the Laravel Installer and how to manage your projects using either the Homestead virtual machine or Valet development environment. We'll also create the companion project which will serve as the basis for introducing new concepts throughout the remainder of the book. I'll also introduce you to several powerful debugging and development tools crucial to efficient Laravel development. Finally, you'll learn a bit about Laravel's automated test environment, and how to write automated tests to ensure your application is operating precisely as expected.

Installing the Laravel Installer

A Laravel package known as the Laravel Installer is indispensable for generating new Laravel project skeletons. The easiest way to install Laravel is via PHP's Composer package manager (https://getcomposer.org). If you're not already using Composer to manage your PHP application dependencies, it's easily installed on all major platforms (OS X, Linux, and Windows among them), so head over to the Composer website and take care of that first before continuing.

With Composer installed, run the following command to install Laravel:

```
1  $ composer global require laravel/installer
```

After installing the Laravel installer, you'll want to add the directory `~/.composer/vendor/bin` to your system path so you can execute the `laravel` command anywhere within the operating system. The process associated with updating the system path is operating system-specific but a quick Google search will produce all of the instructions you need.

With the system path updated, open a terminal and execute the following command:

```
1  $ laravel -V
2  Laravel Installer 1.4.1
```

With that done, let's create the book's companion project skeleton. To generate a Laravel 5.5 project or newer you'll need to be running PHP 7 or newer on your development machine. Also, for reasons that will be apparent later in this chapter, I suggest creating a new directory in your development machine's account home directory named `code`. You don't have to do this, and can certainly manage your Laravel projects anywhere you desire within the file system, however I'll be referring to this directory throughout the next few sections and so it would probably save you some additional thinking to just play along:

```
$ cd ~
$ mkdir code
$ cd code
```

Now that you're inside the `code` directory, let's create the project skeleton. You'll primarily use the `laravel` CLI to generate new Laravel projects, which you can do with the `new` command:

```
$ laravel new hackerpair
Crafting application...
Loading composer repositories with package information
Installing dependencies (including require-dev) from lock file
Package operations: 68 installs, 0 updates, 0 removals
  - Installing doctrine/inflector (v1.2.0): Loading from cache
...
> @php artisan package:discover
Discovered Package: fideloper/proxy
Discovered Package: laravel/tinker
Package manifest generated successfully.
Application ready! Build something amazing
```

If you peek inside the `hackerpair` directory you'll see all of the files and directories which comprise a Laravel application! While I know diving into this code will undoubtedly be a very tantalizing prospect, please be patient and finish reading this chapter in its entirety before getting your hands dirty.

Managing Your Local Laravel Project Hosting Environment

If your development machine is already configured to host PHP applications (and meets a minimum set of requirements itemized here https://laravel.com/docs/master/installation#server-requirements), then you're free to configure your local web server and database to serve the project (use the `public` directory as the project's document root). Even if you've been managing your PHP projects

in this manner for years, I urge you to take this opportunity to at least try one of the local hosting solutions I'll introduce in this chapter.

Like any typical PHP-based web application, Laravel requires a web server such as NGINX or Apache, and in most cases a database such as MySQL or PostgreSQL for hosting application data. Further, modern Laravel applications require PHP 7 or newer. Beyond this, you'll need to update your web server's configuration file to recognize the Laravel application's document root, ensure various required PHP extensions have been installed (see the aforementioned link for a complete set of requirements), and deal with the ongoing system administration-related matters necessary to ensure this software stack plays nicely together. It gets even worse. Your Laravel application may require additional software such as Redis[4] and the npm package manager[5], only adding to the list of third-party technologies you'll have to manage.

In the past dealing with these sorts of distractions was basically a requirement, and along the way a bunch of packaged solutions such as XAMPP (https://www.apachefriends.org/) and MAMP (https://www.mamp.info) were offered as alternatives to manually installing and configuring each part of this stack. In time, a far more convenient and practical solution known as a *virtual machine* came along. A virtual machine is a software-based implementation of a computer that can be run inside the confines of another computer (such as your laptop), or even inside another virtual machine. This is incredible technology, because you can use a virtual machine to run an Ubuntu Linux server on your Windows 10 laptop, or vice versa. Further, it's possible to create a customized virtual machine image preloaded with a select set of software. This image can then be distributed to fellow developers, who can run the virtual machine and take advantage of the custom software configuration. This is precisely what the Laravel developers have done with Homestead[6], a virtual machine which bundles everything you need to get started building Laravel-driven websites.

In this section you'll learn all about Homestead, including how to install and configure it to host your Laravel projects (you can incidentally host all sorts of other PHP projects using Homestead, among them WordPress and Drupal). If you're using OSX, then I recommend you additionally carefully read the subsequent section introducing *Valet*, a streamlined hosting solution which allows you to make new Laravel applications available via your web browser in mere seconds.

Introducing Homestead

Homestead is currently based on Ubuntu 16.04, and includes everything you need to get started building Laravel applications, including PHP 7.1, NGINX, MySQL, PostgreSQL and a variety of other useful utilities such as Redis and Memcached. It runs flawlessly on OS X, Linux and Windows, and the installation process is very straightforward, meaning in most cases you'll be able to begin managing Laravel applications in less than 30 minutes.

[4] http://redis.io/

[5] https://www.npmjs.com/

[6] http://laravel.com/docs/homestead

Installing Homestead

Homestead requires Vagrant[7] and VirtualBox[8] (in lieu of VirtualBox you may use VMware Fusion or Parallels; see the Laravel documentation for more details). User-friendly installers are available for all of the common operating systems, including OS X, Linux and Windows. Take a moment now to install Vagrant and VirtualBox. Once complete, open a terminal and execute the following command:

```
$ vagrant box add laravel/homestead
==> box: Loading metadata for box 'laravel/homestead'
    box: URL: https://vagrantcloud.com/laravel/homestead
This box can work with multiple providers! The providers that it
can work with are listed below. Please review the list and choose
the provider you will be working with.

1) parallels
2) virtualbox
3) vmware_desktop

Enter your choice: 2
==> box: Adding box 'laravel/homestead' (v4.0.0) for provider: virtualbox
    box: Downloading: https://vagrantcloud.com/laravel/boxes/homestead/...
==> box: Successfully added box 'laravel/homestead' (v4.0.0) for 'virtualbox'!
```

Throughout the book I'll use the $ symbol to represent the terminal prompt.

This command installs the Homestead *box*. A box is just a term used to refer to a Vagrant package. Packages are the virtual machine images that contain the operating system and various programs. The Vagrant community maintains hundreds of different boxes useful for building applications using a wide variety of technology stacks, so check out this list of popular boxes[9] for an idea of what else is available.

Once the box has been added, you'll next want to install Homestead. To do so, you'll ideally use Git to clone the repository. If you don't already have Git installed you can easily do so by heading over to the Git website[10] or using your operating system's package manager.

Next, open a terminal and enter your home directory:

[7] http://www.vagrantup.com/
[8] https://www.virtualbox.org/
[9] https://vagrantcloud.com/discover/popular
[10] https://git-scm.com/downloads

```
$ cd ~
```

Then use Git's `clone` command to clone the Homestead repository:

```
$ git clone https://github.com/laravel/homestead.git Homestead
Cloning into 'Homestead'...
remote: Counting objects: 1497, done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 1497 (delta 0), reused 0 (delta 0), pack-reused 1492
Receiving objects: 100% (1497/1497), 241.74 KiB | 95.00 KiB/s, done.
Resolving deltas: 100% (879/879), done.
Checking connectivity... done.
```

If you're not familiar with Git, what you've just done is downloaded the Homestead project repository, which means you not only now possess a copy of the code, but additionally the entire project's history of changes and releases. When you cloned the repository in this fashion, you're currently using what is known as the `master` branch, which may not always be stable. Logically you'll want to use the latest stable release, and so you'll want to `check it out`. At the time of this writing that latest stable release is `v6.5.0`. Check that version out like so:

```
$ cd Homestead
$ git checkout v6.5.0
```

When you'll run this command you'll receive a scary sounding response about being in a "detached head" state. This is irrelevant since you're just going to use Homestead as an end user, so don't worry about it.

You'll see this has resulted in the creation of a directory named `Homestead` in your home directory which contains the repository files. Next, you'll want to enter this directory and execute the following command:

```
$ bash init.sh
Homestead initialized!
```

If you're on Windows you'll instead want to run the following command:

```
$ init.bat
```

Running this script added a few new files to your `Homestead` directory, including `Homestead.yaml`, `after.sh`, and `aliases`. While all three are useful configuration files, for the purposes of just running our newly created project inside the virtual machine we'll only worry about `Homestead.yaml` for now.

Configuring the Homestead.yaml File

Open the `Homestead.yaml` file and you'll find the following contents:

```
---
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.test
      to: /home/vagrant/code/public
```

If you happen to be using VMware Fusion or Parallels then you'll want to update the `provider` property to either `vmware_fusion` or `parallels`, respectively.

Next you'll want to ensure the `authorize` property is pointed to your public SSH key. If you're running Linux or OS X, then chances are high you've generated a public key at some point in the past, and the default `~/.ssh/id_rsa` path is correct. If you're running Linux or OS X and haven't yet generated a key pair then you should be able to do so by running the following command:

```
$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/wjgilmore/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /Users/wjgilmore/.ssh/id_rsa.
Your public key has been saved in /Users/wjgilmore/.ssh/id_rsa.pub.
```

When using keys for reasons of automation, there's really no need to protect the key with a passphrase and so when prompted to provide one just press enter. However, there are very sound reasons for using a passphrase when using keys for other purposes, so be sure to read up on the matter if you're new to key-based authentication!

Windows users don't currently have native key generation capabilities. To my understanding the most straightforward way to generate keys is via the popular PuTTY SSH client. You can learn more about using PuTTY to do so via this link[11].

[11] https://docs.joyent.com/public-cloud/getting-started/ssh-keys/generating-an-ssh-key-manually/manually-generating-your-ssh-key-in-windows

With the `provider` and `authorize` properties sorted, we'll turn attention to the `folders` and `sites` properties. These properties cause quite a bit of confusion among newcomers so pay particular attention to the following explanation.

The `folders` property makes known to the virtual machine the location of one or more applications *residing on your local file system*, and identifies the location on the virtual machine to which these application files should be synchronized. Consider the following default mapping:

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

This means anything residing in a directory named `code` which is found in your home directory will *automatically* be synchronized with the virtual machine's file system, specifically within the directory `/home/vagrant/code`. You're free to synchronize multiple projects by defining additional `map-to` pairs like so:

```
folders:
    - map: ~/code
      to: /home/vagrant/code
    - map: ~/code/hackerpair
      to: /home/vagrant/code/hackerpair
    - map: ~/code/wjgilmore
      to: ~/code/wjgilmore
```

Further, you're not required to use `code` as the local base project directory! For instance if you manage projects in your `Documents/Software` directory, then just change the `folders` property accordingly:

```
folders:
    - map: ~/Documents/software/hackerpair
      to: /home/vagrant/code/hackerpair
    - map: ~/Documents/software/wjgilmore
      to: ~/code/wjgilmore
```

Just keep in mind you don't want to change the `to` reference to the `~/code/` path prefix, because this is where Homestead expects the files to reside. You can actually change this default but there's certainly no reason to do so now.

With your project file system mappings defined, it's time to tell Homestead how the web server should recognize those project directories. This is where the `sites` property comes in. Referring back to the following `folders` configuration:

```
- map: ~/code/hackerpair
  to: /home/vagrant/code/hackerpair
```

We're telling Homestead the `hackerpair` project root directory will be synchronized to `home/vagrant/code/hackerpair`. But this is *not* where a Laravel project's web (also known as *document*) root resides! Laravel project's are always served from the `public` directory, meaning Homestead's web server (known as NGINX) needs to point to that `public` directory when responding to requests. So in the `sites` property you'll want to define the `hackerpair` project like so:

```
folders:
    - map: ~/code/hackerpair
      to: /home/vagrant/code/hackerpair

sites:
    - map: hackerpair.test
      to: /home/vagrant/code/hackerpair/public
```

With these changes in place, you'll be able to reference `http://hackerpair.test` in your browser, and the HackerPair application will be served via Homestead! Not quite, because one minor but important detail remains; you need to tell your local operating system how to resolve references to the `hackerpair.test` domain, because otherwise your browser will reach out to the actual network in an effort to find this site, which in actuality only exists locally.

To ensure proper resolution, you'll need to update your development machine's `hosts` file. If you're running OSX or Linux, this file is found at `/etc/hosts`. If you're running Windows, you'll find the file at `C:\Windows\System32\drivers\etc\hosts`. Open up this file and add the following line:

```
192.168.10.10  hackerpair.test
```

Save these changes, and then run the following command from within your `Homestead` directory:

```
$ vagrant up
Bringing machine 'homestead-7' up with 'virtualbox' provider...
==> homestead-7: Importing base box 'laravel/homestead'...
==> homestead-7: Matching MAC address for NAT networking...
==> homestead-7: Checking if box 'laravel/homestead' is up to date...
==> homestead-7: Setting the name of the VM: homestead-7
==> homestead-7: Clearing any previously set network interfaces...
==> homestead-7: Preparing network interfaces based on configuration...
...
$
```

Your Homestead virtual machine is up and running! With that done, open your browser and navigate to `http://hackerpair.test`. You should see the words "Laravel 5" just as depicted in the following screen shot.

The Laravel 5.5 Splash Screen

Congratulations! From here on out any changes you make to the project will be immediately reflected via the browser. However, there still remains plenty to talk about regarding Homestead and virtual machine management. In the sections that follow I discuss several important matters pertaining to this topic. For the moment I suggest jumping ahead to the section "Perusing the HackerPair Skeleton Code" and returning to the below sections later.

Managing Your Virtual Machine

There are a few administrative tasks you'll occasionally need to carry out regarding management of your virtual machine. For example, if you'd like to shut down the virtual machine you can do so using the following command:

```
1 $ cd ~/Homestead
2 $ vagrant halt
3 ==> homestead-7: Attempting graceful shutdown of VM...
4 $
```

To later boot the machine back up, you can execute `vagrant up` as we did previously:

```
1 $ vagrant up
```

If you'd like to delete the virtual machine (including all data within it), you can use the `destroy` command:

```
$ vagrant destroy
homestead-7: Are you sure you want to destroy the 'homestead-7' VM? [y/N] y
==> homestead-7: Destroying VM and associated drives...
```

I stress executing the `destroy` command this *will delete* not only the virtual machine and also all of its data! Executing this command is very different from shutting down the machine using `halt`. I'm not warning this command will delete your application code, because that is synchronized from your local file system to the virtual machine. It would however delete any data residing in your project databases, since the database is hosted inside the virtual machine.

If you happen to have installed more than one box (it can be addictive), use the `box list` command to display them:

```
$ vagrant box list
laravel/homestead (virtualbox, 4.0.0)
```

Finally, if you make any changes to your `Homestead.yaml` file, you'll need to run the following command in order for Homestead to recognize those changes:

```
$ vagrant reload --provision
```

These are just a few of the many commands available to you. Run `vagrant --help` for a complete listing of what's available:

```
$ vagrant --help
```

SSH'ing Into Your Virtual Machine

Because Homestead is a virtual machine running Ubuntu, you can SSH into it just as you would any other server. For instance you might wish to configure NGINX or MySQL, install additional software, or make other adjustments to the virtual machine environment. If you're running Linux or OS X, you can SSH into the virtual machine using the `ssh` command:

```
$ vagrant ssh
Welcome to Ubuntu 16.04.3 LTS (GNU/Linux 4.4.0-92-generic x86_64)
vagrant@homestead:~$
```

Windows users will need to install an SSH client in order to SSH into the Homestead VM. A popular Windows SSH client is PuTTY[12].

In either case, you'll be logged in as the user `vagrant`, and if you list this user's home directory contents you'll see the `Code` directory defined in the `Homestead.yaml` file:

[12] http://www.putty.org/

```
vagrant@homestead:~$ ls
code
```

If you're new to Linux be sure to spend some time nosing around Ubuntu! This is a perfect opportunity to get familiar with the Linux operating system without any fear of doing serious damage to a server because if something happens to break you can always reinstall the virtual machine.

Transferring Files Between Homestead and Your Laptop

If you create a file on a Homestead and would like to transfer it to your laptop, you have two options. The easiest involves SSH'ing into Homestead and moving the file into one of your shared directories, because the file will instantly be made available for retrieval via your laptop's file system. For instance if you're following along with the `hackerpair` directory configuration, you can SSH into Homestead, move the file into `/home/vagrant/hackerpair`, and then logout of SSH. Then using your local terminal, navigate to `~/code/hackerpair` and you'll find the desired file sitting in your local `hackerpair` root directory.

Alternatively, you can use `sftp` to login to Homestead, navigate to the desired directory, and transfer the file directly:

```
$ sftp -P 2222 vagrant@127.0.0.1
Connected to 127.0.0.1.
sftp>
```

Connecting to Your Database

Although this topic won't really be relevant until we discuss databases in chapter 3, this nonetheless seems a logical place to show you how to connect to your project's Homestead database. If you return to `Homestead.yaml`, you'll find the following section:

```
databases:
    - homestead
```

This section is used to define any databases you'd like to be automatically created when the virtual machine is first booted (or re-provisioned; more about this in the next section). As you can see, a default database named `homestead` has already been defined. You can sign into this database now by SSH'ing into the machine and using the `mysql` client:

```
$ vagrant ssh
Welcome to Ubuntu 16.04.3 LTS (GNU/Linux 4.4.0-92-generic x86_64)
```

After signing in, enter the database using the `mysql` client, supplying the default username of `homestead` and the desired database (also `homestead`). When prompted for the password, enter `secret`:

```
vagrant@homestead:~$ mysql -u homestead homestead -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 5
Server version: 5.7.19-0ubuntu0.16.04.1 (Ubuntu)

Copyright (c) 2000, 2017, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

At this point there are no tables in the database (we'll create a few in chapter 3), but feel free to have a look anyway:

```
mysql> show tables;
Empty set (0.00 sec)
```

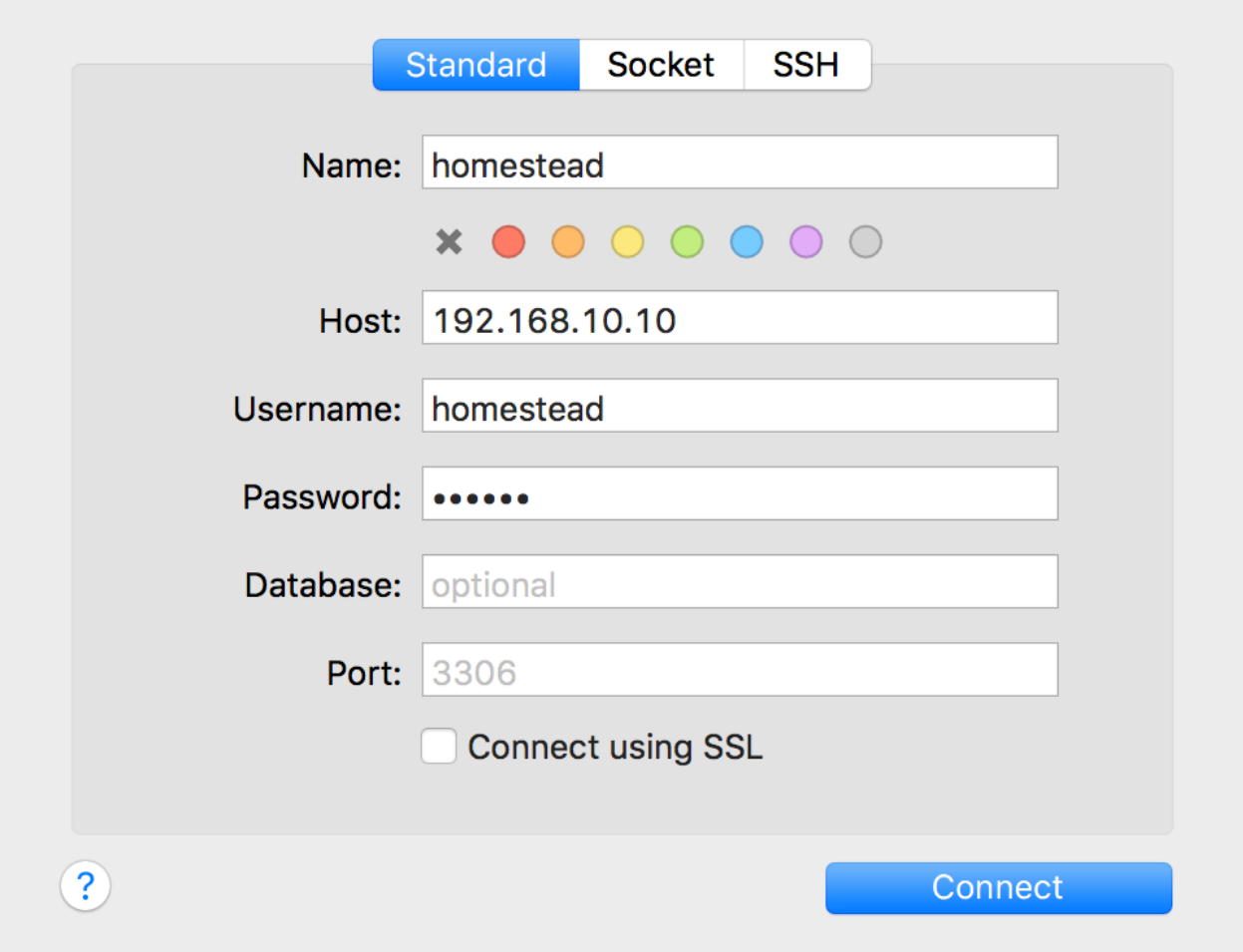
To exit the `mysql` client, execute `exit`:

```
mysql> exit;
Bye
vagrant@homestead:~$
```

Chances are you prefer to interact with your database using a GUI-based application such as Sequel Pro[13] or phpMyAdmin[14]. You'll connect to the `homestead` database like you would any other, by supplying the username (`homestead`), password (`secret`), and the host, which is `192.168.10.10`. For instance, the following screenshot depicts my Sequel Pro connection window:

[13] http://www.sequelpro.com/

[14] https://www.phpmyadmin.net/



The Sequel Pro connection window

You may want to change the name of this default database, or define additional databases as the number of projects you manage via Homestead grows in size. I'll show you how to do this next.

Defining Multiple Homestead Sites and Databases

My guess is you'll quickly become so enamored with Homestead that it will be the default solution for managing all of your Laravel projects. This means you'll need to define multiple projects within the `Homestead.yaml` file. Fortunately, doing so is easier than you think. Check out the following slimmed down version of my own `Homestead.yaml` file, which defines two projects (`hackerpair` and `wjgilmore`):

```
folders:
    - map: ~/code/hackerpair
      to: /home/vagrant/hackerpair
    - map: ~/code/wjgilmore
      to: /home/vagrant/wjgilmore

sites:
    - map: hackerpair.test
      to: /home/vagrant/hackerpair/public
    - map: wjgilmore.test
      to: /home/vagrant/wjgilmore/public

databases:
    - dev_hackerpair
    - dev_wjgilmore
```

Notice how I've also defined two different databases, since each application will logically want its own location to store data.

After saving these changes, you'll want your virtual server to be reconfigured accordingly. If you have never started your virtual server, running `vagrant up` will suffice because the `Homestead.yaml` file had never previously been read. If you've already started the VM then you'll need to force Homestead to *reprovision* the virtual machine. This involves reloading the configuration. To do so, you'll first need to find the identifier used to present the currently running machine:

```
$ vagrant global-status
id       name    provider   state   directory
------------------------------------------------------------------
6f13a59  homestead-7 virtualbox running /Users/wjgilmore/Homestead
```

Copy and paste that `id` value (`6f13a59` in my case), supplying it as an argument to the following command:

```
$ vagrant reload --provision 6f13a59
==> homestead-7: Attempting graceful shutdown of VM...
==> homestead-7: Checking if box 'laravel/homestead' is up to date...
==> homestead-7: Clearing any previously set forwarded ports...
==> homestead-7: Clearing any previously set network interfaces...
==> homestead-7: Preparing network interfaces based on configuration...
...
```

Once this command completes, your latest `Homestead.yaml` changes will be in place!

Introducing Valet

Virtual machines such as Homestead are great, and have become indispensable tools I use on a daily basis. As you've probably gathered from reading the past several pages, Homestead can be overkill for many developers. If you use a Mac and are interested in a no-frills development environment, Laravel offers a streamlined solution called *Valet* which can be configured in mere moments.

Installing Valet

To install Valet you'll need to first install Homebrew (http://brew.sh/), the community-driven package manager for OS X. As you'll see on the home page, Homebrew is very easy to install and should only take a moment to complete. Once done, you'll want to install PHP 7. You can do so by executing the following command:

```
1 $ brew install homebrew/php/php71
```

Next you'll install Valet using Composer. Like Homebrew, Composer (https://getcomposer.org) is a package manager but is specific to PHP development, and is similarly easy to install. With Composer installed, install Valet using the following command:

```
1 $ composer global require laravel/valet
```

Next, add Composer's `bin` directory to your system path. There are a variety of ways to do this but I find the simplest to be editing your home directory's `.bash_profile` file. Open the file in your editor and add the following line to it:

```
1 PATH=$PATH:/~/.composer/vendor/bin
```

Finally, configure Valet by running the following command, which among other things will ensure it always starts automatically whenever you reboot your machine:

```
1 $ valet install
```

Presuming your Laravel applications will use a database, you'll also need to install a database such as MySQL or MariaDB. You can easily install either using Homebrew. For instance, you can install MySQL like so:

```
1 $ brew install mysql
```

After installation completes just follow the instructions displayed in the terminal to ensure MySQL starts automatically upon system boot.

Serving Sites with Valet

With Valet installed and configured, you'll next want to create a directory to host your various Laravel projects. I suggest creating this directory in your home directory; consider calling it something easily recognizable such as `Code` or `Projects`. Enter this directory using your terminal and execute the following command:

```
$ valet park
This directory has been added to Valet's paths.
```

The `park` command tells Valet monitor this directory for Laravel projects, and automatically make a convenient URL available for viewing the project in your browser. For instance, while inside the project directory create a new Laravel project named `hackerpair`:

```
$ laravel new hackerpair
```

Next, if you're using Google Chrome, you'll need to run the following command to change Valet's default use of the `.dev` domain extension to `.test`. I use Chrome for development purposes and so all subsequent URL references will include `.test` however this is just a preference and so you don't need to do this if you're using another browser:

```
$ valet domain test
```

After creating the project, open your browser and navigate to `http://hackerpair.test` and you'll see the project's default splash screen (presented in the following screenshot).

The Laravel splash page

It doesn't get any easier than that!

Perusing the HackerPair Skeleton Code

With the Laravel Installer (and presumably Homestead or Valet) installed and configured, it's time to get our hands dirty! Open a terminal and enter the `hackerpair` project directory. The contents are a combination of files and directories, each of which plays an important role in the functionality of your application so it's important for you to understand their purpose. Let's quickly review the role of each:

- `.env`: Laravel 5 uses the PHP dotenv[15] library to manage your application's configuration variables. You'll use `.env` file as the basis for configuring these settings when working in your development environment. A file named `.env.example` is also included in the project root directory, which should be used as a template from which fellow developers will copy over to `.env` and modify to suit their own needs. I'll talk more about these files and practical approaches for managing your environment settings in the later section, "Configuring Your Laravel Application".
- `.gitattributes`: This file is used by Git[16] to ensure consistent settings across machines, which is useful when multiple developers using a variety of operating systems are working on the same project. You'll find a few default settings in the file; these are pretty standard and you in all likelihood won't have to modify them. Plenty of other attributes are available; Scott Chacon's online book, "Pro Git"[17] includes a section ("Customizing Git - Git Attributes"[18]) with further coverage on this topic.
- `.gitignore`: This file tells Git what files and folders should not be included in the repository. You'll see a few default settings in here, including the `vendor` directory which houses the Laravel source code and other third-party packages, and the `.env` file, which should never be managed in version control since it presumably contains sensitive settings such as database passwords.
- `app`: This directory contains much of the custom code used to power your application, including the models, controllers, and middleware. We'll spend quite a bit of time inside this directory as the book progresses.
- `artisan`: `artisan` is a command-line tool we'll use to rapidly create new parts of your applications such as controllers and models, manage your database's evolution through a great feature known as *migrations*, and interactively debug your application. We'll return to `artisan` repeatedly throughout the book because it is such an integral part of Laravel development.
- `bootstrap`: This directory contains the various files used to initialize a Laravel application, loading the configuration files, various application models and other classes, and define the locations of key directories such as `app` and `public`. Normally you won't have to modify any of the files found in this directory.

[15] https://github.com/vlucas/phpdotenv
[16] http://git-scm.com/
[17] http://git-scm.com/book
[18] http://git-scm.com/book/en/Customizing-Git-Git-Attributes

- `composer.json`: Composer[19] is PHP's de facto package manager, used by thousands of developers around the globe to quickly integrate popular third-party solutions such as Swift Mailer[20] and Doctrine[21] into a PHP application. Laravel heavily depends upon Composer, and you'll use the `composer.json` file to identify the packages you'll like to integrate into your Laravel application. If you're not familiar with Composer by the time you're done reading this book you'll wonder how you ever lived without it. In fact in this introductory chapter alone we'll use it several times to install various useful packages.
- `composer.lock`: This file contains information about the state of your project's installed Composer packages at the time these packages were last installed and/or updated. Like the `bootstrap` directory, you will rarely if ever directly interact with this file.
- `config`: This directory contains several files used to configure various aspects of your Laravel application, such as the database credentials, the cache, e-mail delivery, and session settings.
- `database`: This directory contains the directories used to house your project's database migrations and seed data (migrations and database seeding are both introduced in Chapter 3).
- `package.json`: This file is used to manage locally installed npm (JavaScript) packages. If you look inside the file you'll see references to a number of packages, including jQuery, bootstrap-sass, Laravel Mix, and Vue, among others. Even if you have no JavaScript experience you'll come to find npm packages to be indispensable by the end of this book.
- `phpunit.xml`: Even trivial web applications should be accompanied by an automated test suite. Laravel leaves little room for excuse to avoid this best practice by automatically configuring your application to use the popular PHPUnit[22] test framework. The `phpunit.xml` is PHPUnit's application configuration file, defining characteristics such as the location of the application tests. We'll return to the topic of testing repeatedly throughout the book.
- `resources`: The `resources` directory contains your project's views, localized language files, and raw assets such as Vue components and Sass files.
- `public`: The `public` directory serves as your application's web root directory, housing the `.htaccess`, `robots.txt`, and `favicon.ico` files, in addition to a file named `index.php` that is the *first* file to execute when a user accesses your application. This file is known as the *front controller*, and it is responsible for loading and executing the application. It's because the `index.php` file serves as the front controller that you needed to identify the `public` directory as your application's root directory when configuring `Homestead.yaml` earlier in this chapter.
- `routes`: The `routes` directory is new to 5.3. It replaces the old `app/Http/routes.php` file, and separates your application's routing definitions into four separate files: `api.php`, `channels.php`, `console.php`, and `web.php`. Collectively, these files determine how your application responds to different endpoints. We'll return to these files repeatedly throughout the book, beginning in Chapter 2.
- `server.php`: The `server.php` file can be used to bootstrap your application for the purposes of serving it via PHP's built-in web server. While a nice feature, Homestead and Valet offer a

[19] https://getcomposer.org
[20] http://swiftmailer.org/
[21] http://www.doctrine-project.org/
[22] http://phpunit.de/

far superior development experience and so you can safely ignore this file and feature.
- `storage`: The `storage` directory contains your project's cache, session, and log data.
- `tests`: The `tests` directory contains your project tests. Testing is a recurring theme throughout this book, and thanks to Laravel's incredibly simple test integration features I highly encourage you to follow along closely with the examples provided in these sections.
- `vendor`: The `vendor` directory is where the Laravel framework code itself is stored, in addition to any other third-party code. You won't typically directly interact with anything found in this directory, instead doing so through the Composer interface.
- `webpack.mix.js`: All new Laravel projects include the ability to easily integrate a new feature called *Laravel Mix*. Mix provides a convenient JavaScript-based API for automating various build-related processes associated with your project's CSS, JavaScript, tests, and other assets. I'll introduce Mix in Chapter 2.

Now that you have a rudimentary understanding of the various directories and files comprising a Laravel skeleton application, let's dive a bit deeper into the `config` directory so you have a better understanding of the many different ways in which your application can be tweaked.

Configuring Your Laravel Application

Laravel offers environment-specific configuration, meaning you can define certain behaviors applicable only when you are developing the application, and other behaviors when the application is running in production. For instance you'll certainly want to output errors to the browser during development but ensure errors are only output to the log in production.

Your application's default configuration settings are found in the `config` directory, and are managed in a series of files including:

- `app.php`: The `app.php` file contains settings that have application-wide impact, including whether debug mode is enabled (more on this in a bit), the application URL, timezone, and locale.
- `auth.php`: The `auth.php` file contains settings specific to user authentication, including what model manages your application users, the database table containing the user information, and how password reminders are managed. I'll talk about Laravel's user authentication features in chapter 7.
- `broadcasting.php`: The `broadcasting.php` is used to configure the event broadcasting feature, which is useful when you want to simultaneously notify multiple application users of some event such as the addition of a new blog post. I discuss event broadcasting in chapter 11.
- `cache.php`: Laravel supports several caching drivers, including filesystem, database, memcached, redis, and others. You'll use the `cache.php` configuration file to manage various settings specific to these drivers.

- `database.php`: The `database.php` configuration file defines a variety of database settings, including which of the supported databases the project will use, and the database authorization credentials. You'll learn all about Laravel's database support in chapters 3 and 4.
- `filesystems.php`: The `filesystems.php` configuration file defines the file system your project will use to manage assets such as file uploads. Thanks to Laravel's integration with Flysystem[23], support is available for a wide variety of adapters, among them the local disk, Amazon S3, Azure, Dropbox, FTP, Rackspace, and Redis.
- `mail.php`: As you'll learn in chapter 5 it's pretty easy to send an e-mail from your Laravel application. The `mail.php` configuration file defines various settings used to send those e-mails, including the desired driver (a variety of which are supported, among them Sendmail, SMTP, PHP's `mail()` function, and Mailgun). You can also direct mail to the log file, a technique that is useful for development purposes.
- `queue.php`: Queues can improve application performance by allowing Laravel to offload time- and resource-intensive tasks to a queueing solution such as Beanstalk[24] or Amazon Simple Queue Service[25]. The `queue.php` configuration file defines the desired queue driver and other relevant settings.
- `services.php`: If your application uses a third-party service such as Stripe for payment processing or Mailgun for e-mail delivery you'll use the `services.php` configuration file to define any third-party service-specific settings. We'll return to this file throughout the book as new third-party services are integrated into the application.
- `session.php`: It's entirely likely your application will use sessions to aid in the management of user preferences and other customized content. Laravel supports a number of different session drivers used to facilitate the management of session data, including the file system, cookies, a database, the Alternative PHP Cache, Memcached, and Redis. You'll use the `session.php` configuration file to identify the desired driver, and manage other aspects of Laravel's session management capabilities.
- `view.php`: The `view.php` configuration file defines the default location of your project's view files and the renderer used for pagination.

I suggest spending a few minutes nosing around these files to get a better idea of what configuration options are available to you. There's no need to make any changes at this point, but it's always nice to know what's possible.

Configuring Your Environment

Your application will likely require access to database credentials and other sensitive information such as API keys for accessing third party services. This confidential information should never be shared with others, and therefore you'll want to take care it isn't embedded directly into the

[23] https://github.com/thephpleague/flysystem
[24] http://kr.github.io/beanstalkd/
[25] http://aws.amazon.com/sqs/

code. Instead, you'll want to manage this data within *environment variables*, and then refer to these variables within the application.

Laravel supports a very convenient solution for managing and retrieving these variables thanks to integration with the popular PHP dotenv[26] package. When developing your application you'll define environment variables within the `.env` file found in your project's root directory. The default `.env` file looks like this:

```
APP_NAME=Laravel
APP_ENV=local
APP_KEY=base64:7kPp7zGCLzeXbe0CQuWJ1/lsOymtzZhfmkAUryKyHRF=
APP_DEBUG=true
APP_LOG_LEVEL=debug
APP_URL=http://localhost

DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=homestead
DB_USERNAME=homestead
DB_PASSWORD=secret

BROADCAST_DRIVER=log
CACHE_DRIVER=file
SESSION_DRIVER=file
SESSION_LIFETIME=120
QUEUE_DRIVER=sync

REDIS_HOST=127.0.0.1
REDIS_PASSWORD=null
REDIS_PORT=6379

MAIL_DRIVER=smtp
MAIL_HOST=smtp.mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=null
MAIL_PASSWORD=null
MAIL_ENCRYPTION=null

PUSHER_APP_ID=
PUSHER_APP_KEY=
PUSHER_APP_SECRET=
```

[26] https://github.com/vlucas/phpdotenv

These variables can be retrieved anywhere within your application using the `env()` function. For instance, the `config/database.php` is used to define your project's database connection settings (we'll talk more about this file in chapter 3). It retrieves the `DB_HOST`, `DB_DATABASE`, `DB_USERNAME`, `DB_PASSWORD`, and `DB_SOCKET` variables defined within `.env`:

```
'mysql' => [
    'driver' => 'mysql',
    'host' => env('DB_HOST', '127.0.0.1'),
    'port' => env('DB_PORT', '3306'),
    'database' => env('DB_DATABASE', 'forge'),
    'username' => env('DB_USERNAME', 'forge'),
    'password' => env('DB_PASSWORD', ''),
    'unix_socket' => env('DB_SOCKET', ''),
    'charset' => 'utf8mb4',
    'collation' => 'utf8mb4_unicode_ci',
    'prefix' => '',
    'strict' => true,
    'engine' => null,
],
```

You'll see the `.gitignore` includes `.env` by default. This is because you should *never* manage `.env` in your version control repository! Instead, when it comes time to deploy your application to production, you'll typically define the variables found in `.env` as *server environment variables* which can also be retrieved using PHP's `env()` function. In chapter 9 I'll talk more about managing these variables in other environments.

We'll return to the configuration file throughout the book as new concepts and features are introduced.

Useful Development and Debugging Tools

There are several native Laravel features and third-party tools that can dramatically boost productivity by reducing the amount of time and effort spent identifying and resolving bugs. In this section I'll introduce you to a few of my favorite solutions, and additionally show you how to install and configure the third-party tools.

The debugging and development utilities discussed in this section are specific to Laravel, and do not take into account the many other tools available to PHP in general. Be sure to check out Xdebug[27], and the many tools integrated into PHP IDEs such as Zend Studio[28] and PHPStorm[29].

[27] http://xdebug.org/

[28] http://www.zend.com/en/products/studio

[29] https://www.jetbrains.com/phpstorm/

The dd() Function

Ensuring the `.env` file's `APP_DEBUG` variable is set to `true` is the easiest way to view information about any application errors, because Laravel will dump error- and exception-related information directly to the browser. Sometimes though you'll want to peer into the contents of an object or array even if the data structure isn't causing any particular problem or error. You can do this using Laravel's `dd()`[30] helper function, which will dump a variable's contents to the browser and halt further script execution. For example suppose you defined an array inside a Laravel application and wanted to output its contents to the browser. Here's an example array:

```
$languages = [
    'languages' => [
        'Perl',
        'PHP',
        'Python'
    ]
];
```

You could execute the `dd()` function like so:

```
dd($languages);
```

We haven't yet delved into how to actually add code to your project, so you're probably wondering where this code should go if you want to follow along. Setting up a proper environment for inserting this sort of logic in a natural manner is the subject of another chapter, so for the moment we're going to "cheat" a little and embed the code into our `routes/web.php` routing file. This file is responsible for associating web endpoints (URLs) with corresponding application resources. If you open this file you'll see a single route definition that looks like this:

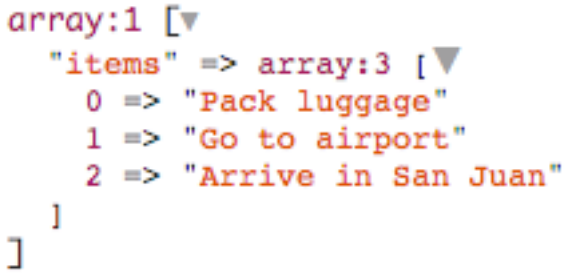
```
Route::get('/', function () {
    return view('welcome');
});
```

This definition ensures that when the application's home page is requested, the template found in `resources/views/welcome.blade.php` is returned. You don't have to understand any of this now because the topic is covered in great detail in the next chapter. For the time being, change this route definition to look like this:

[30] https://laravel.com/docs/master/helpers#method-dd

```
Route::get('/', function () {
    $languages = [
        'languages' => [
            'Perl',
            'PHP',
            'Python'
        ]
    ];
    dd($languages);
    return view('welcome');
});
```

Save the changes and navigate to `http://hackerpair.test` in your browser. Passing `$languages` into `dd()` will cause the array contents to be dumped to the browser window as depicted in the below screenshot.



dd() function output

It is likely at this point you don't know where this code would even be executed. Not to worry! In the chapters to come just keep this and the following solutions in mind so you can easily debug your code once we start building the application.

The Laravel Logger

While the `dd()` helper function is useful for quick evaluation of a variable's contents, taking advantage of Laravel's logging facilities is a more effective approach if you plan on repeatedly monitoring one or several data structures or events without interrupting script execution. Laravel will by default log error-related messages to the application log, located at `storage/logs/laravel.log`. Because Laravel's logging features are managed by Monolog[31], you have a wide array of additional logging options at your disposal, including the ability to write log messages to this log file, set logging levels, send log output to the Chrome console[32] using Chrome Logger[33], or even trigger alerts via e-mail,

[31] https://github.com/Seldaek/monolog
[32] https://developer.chrome.com/devtools/docs/console
[33] http://craig.is/writing/chrome-logger

text messaging, or Slack[34]. Further, if you're using the Laravel Debugbar (introduced later in this chapter) you can easily peruse these messages from the Debugbar's `Messages` tab.

Generating a custom log message is easy, done by embedding one of several available logging methods into the application, passing along the string or variable you'd like to log. Returning to the `$languages` array, suppose you instead wanted to log its contents to Laravel's log:

```
$languages = [
    'languages' => [
        'Perl',
        'PHP',
        'Python'
    ]
];

\Log::debug($languages);
```

After reloading the browser to execute this code, you'll see a log message similar to the following will be appended to `storage/logs/laravel.log`:

```
[2017-11-10 17:59:28] local.DEBUG: array (
  'languages' =>
  array (
    0 => 'Perl',
    1 => 'PHP',
    2 => 'Python',
  ),
)
```

The debug-level message is just one of several at your disposal. Among other levels are info, warning, error and critical, meaning you can use similarly named methods accordingly:

```
\Log::info('Just an informational message.');
\Log::warning('Something may be going wrong.');
\Log::error('Something is definitely going wrong.');
\Log::critical('Danger, Will Robinson! Danger!');
```

[34] https://www.slack.com/

Integrating the Logger and Chrome Logger

When monitoring the log file it's common practice to use the `tail -f` command (available on Linux and OS X; Windows users can use Powershell to achieve a similar behavior) to view any log file changes in real time. You can avoid the additional step of maintaining an additional terminal window for such purposes by instead sending the log messages to the Chrome Logger[35] console, allowing you to see the log messages alongside your application's browser output.

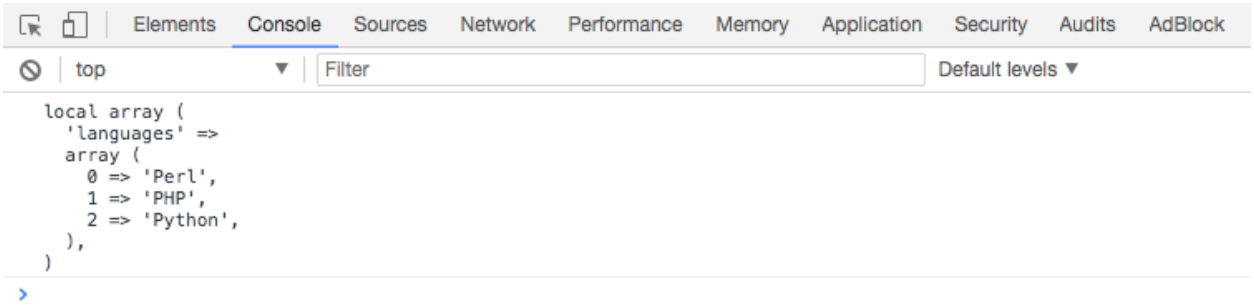
You'll first need to install the Chrome browser and Chrome Logger extension. Once installed, click the Chrome Logger icon once in your browser extension toolbar to enable it for the site. Then add the following anywhere within `bootstrap/app.php`:

```
1  if ($app->environment('local')) {
2      $app->configureMonologUsing(function($monolog) {
3          $monolog->pushHandler(new \Monolog\Handler\ChromePHPHandler());
4      });
5  }
```

You'll want to wrap the configuration logic inside a conditional which ensures your application is running in the local (development) environment, because once deployed you'll want all log messages to be sent to the log file or other third-party logging software. After saving the changes, you can log for instance the `$languages` array just as you did previously:

```
1  \Log::debug($languages);
```

Once executed, the `$languages` array will appear in your browser console as depicted in the below screenshot.



Logging to the Chrome console via the Chrome Logger

[35] http://craig.is/writing/chrome-logger

Using the Tinker Console

You'll often want to test a small PHP snippet or experiment with manipulating a particular data structure, but creating and executing a PHP script for such purposes is kind of tedious. You can eliminate the additional overhead by instead using the tinker console, a command line-based window into your Laravel application. Open tinker by executing the following command from your application's root directory:

```
$ php artisan tinker
Psy Shell v0.8.14 (PHP 7.1.8 â€” cli) by Justin Hileman
>>>
```

Tinker uses PsySH[36], a great interactive PHP console and debugger. PsySH is new to Laravel 5, and is a huge improvement over the previous console. Be sure to take some time perusing the feature list on the PsySH website to learn more about what this great utility can do. In the meantime, let's get used to the interface:

```
>>> $languages = ['Python', 'PHP', 'Perl']
=> [
        "Python",
        "PHP",
        "Perl"
    ]
```

From here you could for instance learn more about how to sort an array using PHP's `sort()` function:

```
>>> sort($languages)
=> true
>>> $languages
=> [
        "Perl",
        "PHP",
        "Python"
    ]
>>>
```

After you're done, type `exit` to exit the PsySH console:

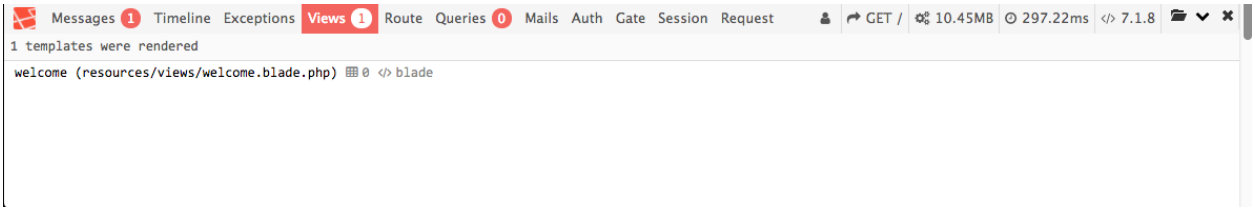
[36] http://psysh.org/

```
>>> exit
Exit:  Goodbye.
$
```

The Tinker console can be incredibly useful for quickly experimenting with PHP snippets, and I'd imagine you'll find yourself repeatedly returning to this indispensable tool. We'll take advantage of Tinker throughout the book to get acquainted with various Laravel features.

Introducing the Laravel Debugbar

It can quickly become difficult to keep tabs on the many different events that are collectively responsible for assembling the application response. You'll regularly want to monitor the status of database requests, routing definitions, view rendering, e-mail transmission and other activities. Fortunately, there exists a great utility called Laravel Debugbar[37] that provides easy access to the status of these events and much more by straddling the bottom of your browser window (see below screenshot).



The Laravel Debugbar

The Debugbar consists of several tabs that when clicked result in context-related information in a panel situated below the menu. These tabs include:

- **Messages**: Use this tab to view log messages directed to the Debugbar. I'll show you how to do this in a moment.
- **Timeline**: Presents a summary of the time required to load the page.
- **Exceptions**: Displays any exceptions thrown while processing the current request.
- **Views**: Provides information about the various views used to render the page, including the layout.
- **Route**: Presents information about the requested route, including the corresponding controller and action.
- **Queries**: Lists the SQL queries executed in the process of serving the request.
- **Mails**: This tab presents information about any e-mails delivered while processing the request.
- **Auth**: Displays information pertaining to user authentication.
- **Gate**: Displays information pertaining to user authorization.

[37] https://github.com/barryvdh/laravel-debugbar

- **Session**: Presents any session-related information made available while processing the request.
- **Request**: Lists information pertinent to the request, including the status code, request headers, response headers, and session attributes.

To install the Laravel Debugbar, execute the following command:

```
$ composer require barryvdh/laravel-debugbar --dev
Using version ^3.1 for barryvdh/laravel-debugbar
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 2 installs, 0 updates, 0 removals
  - Installing maximebf/debugbar (v1.14.1): Downloading (100%)
  - Installing barryvdh/laravel-debugbar (v3.1.0): Downloading (100%)
maximebf/debugbar suggests installing kriswallsmith/assetic
maximebf/debugbar suggests installing predis/predis (Redis storage)
Writing lock file
Generating optimized autoload files
> Illuminate\Foundation\ComposerScripts::postAutoloadDump
> @php artisan package:discover
Discovered Package: fideloper/proxy
Discovered Package: laravel/tinker
Discovered Package: barryvdh/laravel-debugbar
Package manifest generated successfully.
$
```

Save the changes and install the package configuration to your `config` directory:

```
$ php artisan vendor:publish
Copied File [/vendor/barryvdh/laravel-debugbar/config/debugbar.php]
To [/config/debugbar.php]
Publishing complete.
```

While you don't have to make any changes to this configuration file (found in `config/debugbar.php`), I suggest having a look at it to see what changes are available.

Reload the browser and you should see the Debugbar at the bottom of the page! Keep in mind the Debugbar will only render when used in conjunction with an endpoint that actually renders a view to the browser.

The Laravel Debugbar is tremendously useful as it provides easily accessible insight into several key aspects of your application. Additionally, you can use the Messages panel as a convenient location for viewing log messages. Logging to the Debugbar is incredibly easy, done using the Debugbar facade:

```
\Debugbar::error('Something is definitely going wrong.');
```

Save the changes and reload the home page within the browser. Check the Debugbar's `Messages` panel and you'll see the logged message! Like the Laravel logger, the Laravel Debugbar supports the log levels defined in PSR-3[38], meaning methods for debug, info, notice, warning, error, critical, alert and emergency are available.

To disable the Debugbar, you can add the following line of code to the top of your file:

```
\Debugbar::disable();
```

Testing Your Laravel Application

Automated testing is a critical part of today's web development workflow, and should not be ignored even for the most trivial of projects. Fortunately, the Laravel developers agree with this mindset and include support for both PHPUnit and Dusk with every new Laravel project. PHPUnit is a very popular *unit testing framework* which allows you to create well-organized tests used to confirm all parts of your application are working as expected. Dusk is a Laravel sub-project which allows you to test your code *as it behaves in the web browser*, meaning you can ensure your code runs as desired by actually executing it within a browser such as Chrome. Testing is a major theme throughout this book, a subject we'll return to repeatedly to ensure the HackerPair code is correctly implemented. This section kicks things off by getting you acquainted with Laravel's default test infrastructure and PHPUnit fundamentals.

Introducing Unit Tests

Each new Laravel application even includes two example tests which you can use as a reference for beginning to write your own tests! One of the tests is located inside the `tests/Unit` directory. Tests placed in this directory are intended to ensure the smallest possible units of code are behaving as desired. For instance in later chapters we'll write unit tests to ensure model instance methods are returning correct output in conjunction with a variety of circumstances. The default test found in the `Unit` directory is named `ExampleTest.php` and it looks like this:

[38] http://www.php-fig.org/psr/psr-3/

```php
<?php

namespace Tests\Unit;

use Tests\TestCase;
use Illuminate\Foundation\Testing\RefreshDatabase;

class ExampleTest extends TestCase
{
    /**
     * A basic test example.
     *
     * @return void
     */
    public function testBasicTest()
    {
        $this->assertTrue(true);
    }
}
```

Granted this isn't much to look at, but even so it gives you an idea of the basic test structure. Tests are managed within classes, with each test encapsulated in a class method. Each test begins with the prefix `test`, and will contain one or more *assertions*. Assertions intend to confirm your code's conformance to a requirement. For instance, you might assert that a method response value is `true`, `false`, `New York City`, or null.

Taking these characteristics into consideration, the `ExampleTest` class contains a single test named `testBasicTest`. It uses Laravel's testing API to interact with PHPUnit, confirming that the value `true` does in fact equal true (`assertTrue(true)`). You can run this test by executing the following command:

```
$ vendor/bin/phpunit tests/Unit/ExampleTest.php
PHPUnit 6.4.3 by Sebastian Bergmann and contributors.

.                                                   1 / 1 (100%)

Time: 242 ms, Memory: 10.00MB

OK (1 test, 1 assertion)
```

That period is indicative of a passing test. You're also told how many tests and assertions ran (`1 test, 1 assertion`). Let's add another passing test to the class:

```
public function testSomeValueIsFalse()
{
    $this->assertFalse(false);
}
```

Run the test suite again to see the results:

```
$ vendor/bin/phpunit tests/Unit/ExampleTest.php
PHPUnit 6.4.3 by Sebastian Bergmann and contributors.

..                                  2 / 2 (100%)

Time: 200 ms, Memory: 10.00MB

OK (2 tests, 2 assertions)
```

Unfortunately, your tests will rarely pass on the first time; that's just part of writing code. To see what a failing test looks like, change the `assertTrue` method in `testBasicTest` to look like this:

```
$this->assertTrue(false);
```

Run the test anew and you'll see an `F` in place of the period, and some feedback regarding why the test failed:

```
$ vendor/bin/phpunit tests/Unit/ExampleTest.php
PHPUnit 6.4.3 by Sebastian Bergmann and contributors.

F                                   1 / 1 (100%)

Time: 195 ms, Memory: 10.00MB

There was 1 failure:

1) Tests\Unit\ExampleTest::testBasicTest
Failed asserting that false is true.

/Users/wjgilmore/Code/valet/hackerpair/tests/Unit/ExampleTest.php:17

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

In particular, note the reference to the line number causing the failed assertion (17). This feedback will be very useful in terms of helping you to quickly track down why your tests are failing, and in later chapters I'll introduce even more efficient solutions when the reason isn't so obvious.

`assertTrue` and `assertFalse` are just two of many available assertion methods. For instance you would use `assertEquals` to confirm that a particular return value matches expectations:

```
public function testUserFullNameIsJasonGilmore()
{
    $fullName = "Jason Gilmore";
    $this->assertEquals("Jason Gilmore", $fullName);
}
```

You would use `assertCount` to confirm an array contains the expected number of values:

```
public function testUserHasFavoritedFiveEvents()
{
    $favorites = [45, 12, 676, 88, 15];
    $this->assertCount(5, $favorites);
}
```

Creating Your Own Test

You can easily create a test skeleton using the following command:

```
$ php artisan make:test TicketsTest --unit
```

This will create a new test inside the `tests/Unit` directory named `TicketsTest.php`. Open it up and you'll find the following contents:

```
<?php

namespace Tests\Unit;

use Tests\TestCase;
use Illuminate\Foundation\Testing\RefreshDatabase;

class TicketsTest extends TestCase
{
    public function testExample()
    {
        $this->assertTrue(true);
    }
}
```

Once generated you can go about modifying (or deleting) the example test. We'll return to unit tests repeatedly throughout the book, introducing other assertion methods along the way. In the meantime, browse the PHPUnit documentation[39] for a preview of what's available.

Introducing Feature Tests

Another example test is found in the directory `tests/Feature` and also named `ExampleTest.php`. Feature tests are intended to confirm the behavior of multiple code units working together and often involve performing HTTP requests. In fact, the example test issues an HTTP request to retrieve the project home page, and determines whether a 200 response status code is returned (a 200 status code is indicative of a successful request):

```
<?php

namespace Tests\Feature;

use Tests\TestCase;
use Illuminate\Foundation\Testing\RefreshDatabase;

class ExampleTest extends TestCase
{
    /**
     * A basic test example.
     *
     * @return void
     */
    public function testBasicTest()
    {
        $response = $this->get('/');

        $response->assertStatus(200);
    }
}
```

To run the test, execute the `phpunit` command from within your project's root directory:

[39] https://phpunit.de/manual/current/en/appendixes.assertions.html

```
$ vendor/bin/phpunit tests/Feature/ExampleTest.php
PHPUnit 6.4.3 by Sebastian Bergmann and contributors.

.                                                   1 / 1 (100%)

Time: 130 ms, Memory: 12.00MB

OK (1 test, 1 assertion)
```

As you learned earlier in the chapter, the `web.php` routes file only contains a single route definition pointing to `/`. Let's confirm the `/contact` URI doesn't exist by attempting to retrieve it and confirming a 404 status code is returned:

```
public function testNonexistentEndpointReturns404()
{
    $response = $this->get('/contact');

    $response->assertStatus(404);
}
```

You can test much more than mere status codes; for instance you can use the `assertSeeText` method to determine whether the string `Laravel` is found in the response:

```
public function testHomepageContainsProjectName()
{
    $response = $this->get('/');

    $response->assertSeeText('Laravel');
}
```

Like unit tests, we'll return to feature tests throughout the book, and later we'll more heavily rely upon a relatively recent Laravel testing solution known as Dusk whenever the test involves interaction with web page elements. Despite Laravel Dusk's rise to prominence (Dusk is introduced in chapter 2), feature tests certainly continue to play an important role in ensuring application quality, and in later chapters we'll return to them whenever appropriate.

In the meantime, have a look at the Laravel documentation[40] for a list of assertions which can be used in conjunction with the `get`, `post`, `put`, `delete`, and `json` methods (I'll introduce these other test helper methods in later chapters).

[40] https://laravel.com/docs/5.5/http-tests

Bear in mind that while the `get` method is in fact retrieving your project's home page via an HTTP request, this does not involve a web browser. Therefore any JavaScript which may execute on a given endpoint is not going to execute, possibly resulting in unexpected results. You can test JavaScript within automated tests using Laravel Dusk, a Laravel feature we'll explore in chapter 2.

Additional Testing Resources

Automated testing is such an important part of building modern web applications that you owe it to yourself, your employer, and your clients to incorporate it into all of your projects. In doing so, you'll save untold amounts of time, pain, and money, not to mention allow you to focus on the entertaining aspects of web development rather than dreary manual testing and bug hunting. That said, I encourage you to keep the following resources in mind as you continue reading this book:

- Almost every chapter in this book concludes with a section explaining how to test the chapter's subject matter.
- Chapter 2 introduces Laravel Dusk, an amazing integration testing solution which allows you to confirm how your web application runs inside an actual browser. This capability is particularly crucial for applications which include JavaScript, since JavaScript is otherwise not capable of being tested using the other automated approaches discussed in this book.
- Laracasts (https://laracasts.com/) includes a free video series called "Testing Laravel" which covers Laravel testing fundamentals.
- Adam Wathan's Test-Driven Laravel[41] is undoubtedly the reference resource for learning how to test all facets of Laravel applications.

Conclusion

It's only the end of the first chapter and we've already covered a tremendous amount of ground! With your project generated and development environment configured, it's time to begin building the application. Onwards!

[41] http://www.testdrivenlaravel.com

Chapter 2. Managing Your Project Controllers, Layout, Views, and Other Assets

The typical dynamic web page consists of various components which are assembled at runtime to produce what the user ultimately sees in the browser. These components include the *view*, which consists of the design elements and content specific to the requested page, the *layout*, which consists of the page header, footer, and other design elements that generally globally appear throughout the site, and other assets such as the images, JavaScript and CSS. Web frameworks such as Laravel create and return these pages by routing requests for a specific page to an associated *controller* and *action*.

This chapter shows you exactly how this works, and how to marry your project's controllers and actions to *routes* such as `http://hackerpair.com/events`. You'll also learn how to create a project layout and views, and how to use Laravel Mix to automate the management of otherwise tedious tasks such as CSS and JavaScript compilation. I'll also show you how to integrate Bootstrap 3 and Bootstrap 4. We'll conclude the chapter with several examples demonstrating how to test your views, controllers, and JavaScript.

Creating Your First View

In the previous chapter we created the example project and viewed the default landing page within the browser. The page was pretty sparse, consisting of the text "Laravel 5". If you view the page's source from within the browser, you'll see a few CSS styles are defined, a Google font reference, and some simple HTML. You'll find this view in the file `welcome.blade.php`, found in the directory `resources/views`. Open this file in your PHP editor, and update it to look like this:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Welcome to HackerPair</title>
  </head>
  <body>
  <h1>Welcome to HackerPair</h1>
  </body>
</html>
```

Reload the application's home page within the browser, and you should see the header `Welcome to HackerPair`. Congratulations! You've just created your first Laravel view. It looks pretty lame though, and the Chrome browser's default Times New Roman font isn't exactly adding to the allure, but no matter; we'll improve the look soon enough.

So why is this particular view returned when you navigate to the home page? The `welcome.blade.php` view is served by a *route definition* found in the `routes/web.php` file:

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

This code block tells Laravel to serve the `welcome.blade.php` file when a GET request is made to the application's homepage, represented by the forward slash (`/`). The majority of your views will be served in conjunction with a controller (more about this in a bit), but if some particular page contains purely static content (such as an "About Us" page) then the above approach is a perfectly acceptable solution.

Because Laravel presumes all views will use the `blade.php` extension (you'll learn more about the meaning of `blade` later in this chapter), you're spared the hassle of including `.blade.php` in the view declaration. Also, you're free to name the view prefix whatever you please; for instance try renaming `welcome.blade.php` to `hello.blade.php`, and then update the `view` function to look like this:

```
return view('hello');
```

Reload the browser and you'll see the same outcome as before, despite the filename change.

For organizational purposes you can manage views in separate directories. To do so you can use a convenient dot-notation syntax for representing the directory hierarchy. For instance you could organize views according to controller by creating a series of aptly-named directories in `resources/views`. As an example, create a directory named `welcome` in `resources/views`, move `welcome.blade.php` into this directory, and rename the file to `index.blade.php`. Then update the route to look like this:

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

You're certainly not required to manage views in this fashion, however I find the approach indispensable given that a typical Laravel application can quickly grow to include dozens, if not hundreds, of views.

Incidentally, connecting endpoints to static content is so commonplace that in Laravel 5.5 a new route method was added which streamlines the route definition:

```
Route::view('/', 'welcome');
```

Feel free to try swapping out the original definition for the streamlined version and reloading the home page just to see it in action.

Creating Your First Controller

The lone default route serves the important purpose of giving you *something* to see when accessing the home page of a newly generated Laravel application, but in practice only a few of your application's routes will serve static content. This is because the majority of your views will contain some degree of dynamic data, and this dynamic data will be retrieved and passed into a view by way of a *controller*.

We're not yet ready to begin passing dynamic data into a view (I'll introduce this topic later in the chapter), however it seems a fine time to learn how to create a controller capable of serving the welcome view. You can easily generate controllers using Artisan's `make:controller` command:

```
$ php artisan make:controller WelcomeController
Controller created successfully.
```

When generating controllers with `make:controller`, Laravel will by default create an empty class devoid of any properties or methods. The controller files will be placed in `app/Http/Controllers`, and assigned whatever name you supplied as an argument (in the case of the above example, the name will be `WelcomeController.php`). The generated class file looks like this:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class WelcomeController extends Controller
{
    //
}
```

With the controller generated, let's create an `index` action and corresponding view. Open the newly created controller (`app/Http/Controllers/WelcomeController.php`) and add the following `index` method to the class:

```
function index()
{
    return view('welcome.index');
}
```

Controller class methods which respond to an application endpoint request are often referred to as *actions*. I'll refer to these methods as `endpoints` whenever applicable.

This example presumes you'd like to adhere to the best practice discussed earlier pertaining to organizing views within subdirectories. To do so, create a new directory inside `resources/views` named `welcome`, and move the `welcome.blade.php` view currently found in the `resources/views` directory into the newly created `welcome` directory.

After saving these changes, open the `web.php` file (`routes/web.php`). Replace the lone defined route with the following route:

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

This route tells Laravel to respond to GET requests made to the application home page (`/`) by executing the `WelcomeController`'s `index` action. Save these changes, return to the browser, and reload the home page.

By the way, presuming you've installed the Laravel Debugbar, now would be an ideal time to have a look at the `Views` and `Route` tabs. You'll see the former plainly identifies the view file served in association with this request, and the latter includes all sorts of useful information about the route, including the URI (`GET /`), the controller (`App\Http\Controllers\WelcomeController@index`), and even specific references to lines of code in the controller which executed in conjunction with the request (`10-13`).

Defining Route Parameters

Many route URIs include dynamic data which is used to form the response. For instance one of HackerPair's primary responsibilities involves displaying information about various events taking place around the country. If you navigate to `http://hackerpair.dev/events`, click on any of the events listed there, and then have a look at the destination URL it will look something like this:

```
http://hackerpair.dev/events/42
```

The number `42` is a unique ID associated with a particular event. This approach is commonplace when the need arises to retrieve data found in an application database, and that unique ID is typically a table record's primary key. But I'm getting ahead of myself, as database integration is a topic addressed in great detail in the next chapter. For the moment, just consider that the controller associated with this endpoint needs to somehow retrieve that value in order to do *something* with it. Let's put what you're about to learn into practice by first creating a new controller named `EventsController`:

```
$ php artisan make:controller EventsController
```

Now open this controller and add an action named `show`:

```
public function show($id)
{
    dd($id);
}
```

Notice how a variable named `$id` is passed into the action? The action body will then just use Laravel's `dd` helper (dump and die) to output the variable's value. Now navigate to `http://hackerpair.dev/events/42` to see the result.

Aha, a 404 error was returned. That's because we still need to define the route inside `web.php`! To pass this value into a controller action you'll need to define the route like so:

```
Route::get('events/{id}', 'EventsController@show');
```

With this crucial step complete, open your browser and navigate to `http://hackerpair.dev/events/42`. After loading the page, you should see `"42"` output to the browser window.

Experienced readers might be wondering whether Laravel supports RESTful controllers, negating the need to manually create these sorts of fundamental endpoints. Indeed it does, and REST is an integral part of both the HackerPair project and the rest of this book as of chapter 3.

Defining Multiple Route Parameters

If you need to pass along multiple parameters just specify them within the route definition as before:

```
Route::get(
  'events/category/{category}/{subcategory}',
  'EventsController@category'
);
```

Then in the corresponding action be sure to define the input arguments in the same order as the parameters are specified in the route definition:

```
public function category($category, $subcategory)
{
  dd("Category: {$category} Subcategory: {$subcategory}");
}
```

With this in place, navigate to `http://hackerpair.dev/events/category/php/laravel` and you should see the following string output to your browser:

```
"Category: php Subcategory: laravel"
```

Optional Route Parameters

All of the route parameters we've defined thus far are *required*. Neglecting to include them in the URL will result in an error. For instance navigating to `http://hackerpair.dev/events/category/php` will produce a 404 error, because Laravel doesn't recognize a route matching `events/category/{category}`. Sometimes you'll want to define these parameters as *optional*, and assign default values should the optional parameter(s) not be provided. To define an optional parameter you'll append a question mark onto the parameter name, like this:

```
Route::get(
  'events/category/{category}/{subcategory?}',
  'EventsController@category'
);
```

You can then identify the parameter is optional in the associated action by defining it as optional in the method declaration and then checking and responding accordingly to the value in the method body:

```
public function category($category, $subcategory = 'all')
{
  dd("Category: {$category} Subcategory: {$subcategory}");
}
```

Note how I assigned a default value of `all` to the `$subcategory` input parameter. If no value is provided via the route URI, then `$subcategory` will automatically be assigned `all` within the action body. Try this out by navigating to `http://hackerpair.dev/events/category/php`, then `http://hackerpair.dev/events/category/php/laravel`, and note the difference in output.

Assigning Route Names

If you've any web development experience at all then you're quite familiar with the HTML anchor element (the `a` tag), used to create hyperlinks from one page (or website) to the next:

```
<a href="/events/42">Laravel Hacking and Coffee</a>
```

The typical Laravel-powered web application will naturally contain all sorts of hyperlinks, yet those of you who are being introduced to web framework-driven development might be surprised to know that most of these links are generated rather than hand coded. This is because a concept known as using *named routes* is widely considered to be a best practice. Using named routes, we never actually have to explicitly refer to an endpoint's URI within a view, and instead can generate the URI using the route name. Returning to the earlier route definition in which we display event information based on its ID, let's modify the definition to include a name:

```
Route::get('events/{id}', 'EventsController@show')->name('events.show');
```

Once defined, we can generate the `/events/{id}` URI within views using Laravel's `route` helper. Add the following code to your `welcome/index.blade.php` body:

```
<p>
  <a href="{{ route('events.show', ['id' => 42]) }}">Laravel Hacking and Coffee<\
/a>
</p>
```

Reload the browser and you'll see that a hyperlink which looks like this has been generated:

```
<a href="http://hackerpair.dev/events/42">Laravel Hacking and Coffee</a>
```

Newcomers to this concept might be left scratching their head. Why is this useful? Why not just hand code the URLs since we're presumably hand coding a bunch of other view-specific content? Imagine you've in fact hand-coded the event URIs, and now they're found throughout the HackerPair application. After presenting the new feature to the team, the consensus is that HackerPair events should instead be called `meetups`, necessitating a change to all of these URIs. Now you're going to subsequently be stuck performing a find and replace in order to refactor the code. If you'd instead used named routes, the change would be limited to a single line of code found in your `web.php` file:

```
Route::get('meetups/{id}', 'EventsController@show')->name('events.show');
```

Another reason to avoid hand-coded URLs is because your Laravel application's will typically display a great deal of data retrieved from a database. This means much of the page content will be displayed using PHP code rather than raw HTML (using a special syntax called Blade which I'll introduce in just a bit). Because you'll spend so much time using this display logic, your code can become incredibly unreadable if you're constantly switching in and out of raw HTML:

```
<ul>
  @foreach ($events as $event)
    <li><a href="/events/{{ $event->id}}">{{ $event->name }}</a></li>
  @endforeach
</ul>
```

Don't sweat the references to `@foreach` and the curly brackets for right now; they're part of the aforementioned Blade templating language. What's important is how unwieldly the above code is when compared to using route names for URL generation:

```
<ul>
  @foreach ($events as $event)
    <li>
      <a href="{{ route('events.show', ['id' => $event->id]) }}">
        {{ $event->name}}
      </a>
    </li>
    @endforeach
</ul>
```

Personally I find hand-coding even the `<a href=""></a>` to be annoying. Later in this chapter I'll introduce you to a powerful package named Laravel Collective package which supports another helper named `link_to_route`. This helper allows you to to programmatically generate the entire URL:

```
<ul>
  @foreach ($events as $event)
    <li>
      {!! link_to_route('events.show', $event->name, ['id' => $event->id]) !!}
    </li>
  @endforeach
</ul>
```

Managing Your Application Routes

As of Laravel 5.3, the route file previously found in `app/Http/routes.php` has been split into multiple files, all of which are now found in the `routes` directory. This was done to accommodate the different kinds of routes which might accompany a modern web application. For instance, in addition to a public-facing website, your Laravel project might also offer an API. Further, you might create a console-based interface which might be used to conveniently carry out various administrative tasks

such as backing up the database or assembling and displaying useful statistics. This is why in that `routes` directory you'll find `api.php`, `channels.php`, `console.php`, and `web.php` files; each is tasked with managing routes associated with specific parts of your application, namely API, broadcast events, console, and web.

So why separate these three sets of routes in the first place? Notably because web-related routes will likely be used in conjunction with features such as cookie-based authentication and persistent user sessions, something not needed when building an API. Also, juggling both API- and web-related routes in the same routes file can get downright annoying, with some applications supporting several hundred routes.

In this chapter you've already created a few routes in the `web.php` configuration file, and accessed those routes via the browser. Indeed we'll spend much of the rest of this book focused on web-related routing, and at least some of what you'll learn in this regards will be relevant for API development. In chapter 13 you'll use the `api.php` file in conjunction with creating a HackerPair API. That said, much of the rest of this section covers material of relevance to web and API developers alike, so pay careful attention!

Listing Routes

As your application progresses it can be easy to forget details about the various route combinations. You can view a list of all available routes using the `php artisan route:list` command. The output of this command has grown to the point where displaying an example here is impossible due to space restrictions, so instead I invite you to execute the command and I'll explain the meaning of each column:

- `Domain`: It's possible to define sub-domain routes in your route files. If a sub-domain is associated with a given route, its name will be displayed here.
- `Method`: The HTTP method used to contact the endpoint, for example `GET`, `POST`, or `DELETE`.
- `URI`: The string used to represent a network resource. For instance, in previous examples we've been referring to the URI `events/{id}`.
- `Name`: The name of the route alias. When working with resourceful controllers (resourceful controllers are introduced in the next chapter) Laravel will provide default aliases. For instance, a resourceful `Events` controller's `show` action's default route alias is `events.show`. If you're not using a resourceful controller, you can define custom route aliases as described in the earlier section, "Creating Route Names".
- `Action`: The controller action associated with the route. For instance, the `Events` controller's `index` action will be identified here as `EventsController@show`.
- `Middleware`: Laravel supports filtering route requests using middleware. If a route is affected by one or more middleware, those middleware names will be listed here. You'll learn all about route middleware in Chapter 6.

Introducing the Blade Template Engine

One of the primary goals of an MVC framework such as Laravel is *separation of concerns*. We don't want to pollute views with database queries and other logic, and likewise don't want the controllers and models to determine how data should be presented. Because the views are intended to be largely devoid of any programming language syntax, they can be easily maintained by a designer who might lack programming experience.

But certainly *some* logic must be found in the view, otherwise how would we loop over an array to create a table? Or present a visual cue if the user has favorited an event? To achieve a happy medium, most frameworks provide a simplified syntax for embedding logic into a view. Such facilities are known as *template engines*. Laravel's template engine is called *Blade*. Blade offers all of the features one would expect of a template engine, including inheritance, output filtering, and conditional and looping statements.

Laravel uses the `.blade.php` extension to identify Blade-augmented views. In this section we'll work through a number of different examples demonstrating the many capabilities of this powerful syntax.

Displaying Variables

Your views will typically include dynamic data passed to it from the corresponding controller action. For instance, suppose you wanted to pass the name of a property retrieved from the database into a view. For the purposes of demonstration we'll use the previously created `Events` controller's `show` action (if you want to follow along be sure this controller, action, view, and associated route are in place):

```
1  public function show($id)
2  {
3
4      return view('events.show')
5          ->with('id', $id);
6
7  }
```

This action will look for a view named `show.blade.php` found in the `resources/views/events` directory, so go ahead and create that file now. Then add the following code to that file:

```
1  {{-- Output the $id variable. --}}
2  <p>We're looking at event ID #{{ $id }}.</p>
```

The first line presents an example of a Blade comment. Blade comments are enclosed within the `{{--` and `--}}` tags, and will be removed from the rendered web page.

The second line references the variable name. Variables are referenced in Laravel views in the same fashion as you would see in any PHP script. The variable is additionally enclosed within curly brackets so Laravel can differentiate these variables from other parts of the page content.

After saving the changes, reload `http://hackerpair.dev/events/42` and you should see `We're looking at event ID #42.` embedded into the view!

You can optionally use a shortcut to identify the variable name:

```
1  public function show($id)
2  {
3
4      return view('events.show')
5          ->withId($id);
6
7  }
```

This variable is then made available to the view exactly as before:

```
1  <p>We're looking at event ID #{{ $id }}.</p>
```

Incidentally, I prefer to include a space between the curly brackets and variable for readability's sake, but you're not required to do so. Laravel will parse the `$name` variable in identical fashion:

```
1  <p>We're looking at event ID #{{$id}}.</p>
```

Displaying Multiple Variables

You'll certainly want to pass multiple variables into a view. The easiest way involves invoking the `with` method multiple times:

```
public function show($id)
{

    return view('events.show')
        ->with('id', $id)
        ->with('name', 'Laravel Hacking and Coffee');

}
```

Then in the `events.show` (`resources/views/events/show.blade.php`) view you can use both variables just as you did previously when only `$id` was passed in:

```
<p>
    {{ $name }} has the event ID #{{ $id }}.
</p>
```

Logically this latter approach could get rather unwieldy if you needed to pass along more than two variables. Save some typing by using PHP's `compact()` function:

```
$name = 'Hilton Head Beach House';
$date = date('Y-m-d');
return view('welcome', compact('name', 'date'));
```

The `$name` and `$date` variables defined in your action will then automatically be made available to the view. If this is confusing see the PHP manual's `compact()` function documentation at http://php.net/compact[42].

If you need to pass along a larger number of variables, then repeatedly invoking `with` can get annoying. Consider packaging those variables into an array instead:

```
public function show($id)
{

    $data = [
        'name' => 'Laravel Hacking and Coffee',
        'date' => date('Y-m-d')
    ];

    return view('events.show')->with($data);

}
```

To display the `$name` and `$date` variables within the view, just update your view to reference both:

[42] http://php.net/compact

```
{{ $name }} is scheduled for {{ $date }}!
```

Confirming Variable Existence

There are plenty of occasions when a particular variable might not be set at all, and if not you want to output a default value. Consider a future feature where registered HackerPair members have the option of specifying a name, but not all members choose to do so. In cases where a name isn't available you'll display a default moniker:

```
Welcome, {{ $name or 'HackerPair Member' }}!
```

Escaping Dangerous Input

Because web applications often display user-contributed data (e.g. product reviews and blog comments), you must take great care to ensure malicious data isn't inserted into the database. You'll typically do this by employing a multi-layered filter, starting by properly validating data (discussed in Chapter 3) and additionally escaping potentially dangerous data (such as JavaScript code) prior to embedding it into a view. In Laravel 5 you'll use the `{!!` and `!!}` delimiters to output raw data:

```
{!! 'My list <script>alert("spam spam spam!")</script>' !!}
```

You should only output raw data (using `{{ ... }}`) when you're certain it does not originate from a potentially dangerous source, which frankly is almost never the case.

Looping Over an Array

HackerPair users spend a lot of time perusing events, posting their own events, and updating profiles. You'll want to display lists of various sorts within the views, and Blade supports several constructs for doing so. Let's demonstrate this concept by iterating over an array of properties in the `Events` controller's `index` view. Begin by adding the following action to your `Events` controller:

```
public function index()
{

}
```

Next, open `routes/web.php` and add the following line to the file:

```
Route::get('events', 'EventsController@index');
```

and then modify the action to look like this:

```
public function index()
{
    $events = [
        'Laravel Hacking and Coffee',
        'IoT with Raspberry Pi',
        'Free Vue.js Lessons'
    ];
    return view('events.index')->with('events', $events);
}
```

Next, update the `index` view to include the following code:

```
<ul>
  @foreach ($events as $event)
    <li>{{ $event }}</li>
  @endforeach
</ul>
```

When rendered to the browser (`http://hackerpair.dev/events`), the following HTML is produced:

```
<ul>
    <li>Laravel Hacking and Coffee</li>
    <li>IoT with Raspberry Pi</li>
    <li>Free Vue.js Lessons</li>
</ul>
```

Because the array could be empty, consider checking its size before iterating over it. You could use an `@if` conditional (introduced next) combined with the `count()` function to do so, or save a few keystrokes by using the `@forelse` construct instead to display an appropriate message should no array items exist:

```
<ul>
  @forelse ($events as $event)
    <li>{{ $events }}</li>
  @empty
    <li>No events available.</li>
  @endforelse
</ul>
```

This variation will iterate over the `$events` array just as before. If the array happens to be empty the block of code defined in the `@empty` directive will instead be executed.

If Conditional

In the previous example I introduced the `@forelse` directive. While useful, for readability reasons I'm not personally a fan of this syntax and instead use the `@if` directive to determine array size:

```
<ul>
  @foreach ($events as $event)
    <li>
        {{ $event }}
        @if (strpos($event, 'Laravel') !== false)
         (sweet framework!)
        @endif
    </li>
  @endforeach
</ul>
```

Reloading the page produces the following HTML:

```
<ul>
    <li>Laravel Hacking and Coffee (sweet framework!)</li>
    <li>IoT with Raspberry Pi</li>
    <li>Free Vue.js Lessons</li>
</ul>
```

Blade also supports the if-elseif-else construct:

```
<ul>
  @foreach ($events as $event)
    <li>
        {{ $event }}
        @if (strpos($event, 'Laravel') !== false)
         (sweet framework!)
        @elseif (strpos($event, 'Raspberry') !== false)
         (love me some Raspberry Pi!)
        @else
         (don't know much about this one!)
        @endif
    </li>
  @endforeach
</ul>
```

Reloading the page produces the following HTML:

```
<ul>
    <li>Laravel Hacking and Coffee (sweet framework!)</li>
    <li>IoT with Raspberry Pi(love me some Raspberry Pi!)</li>
    <li>Free Vue.js Lessons(don't know much about this one!)</li>
</ul>
```

Managing Your Application Layout

The typical web application consists of a design elements such as a header and footer, and these elements are generally found on every page. Because eliminating redundancy is one of Laravel's central tenets, clearly you won't want to repeatedly embed elements such as the site logo and navigation bar within every view. Instead, you'll use Blade to create a *master layout* that can then be inherited by the various page-specific views. To create a layout, first create a directory within `resources/views` called `layouts`, and inside it create a file named `app.blade.php`. Add the following contents to this newly created file:

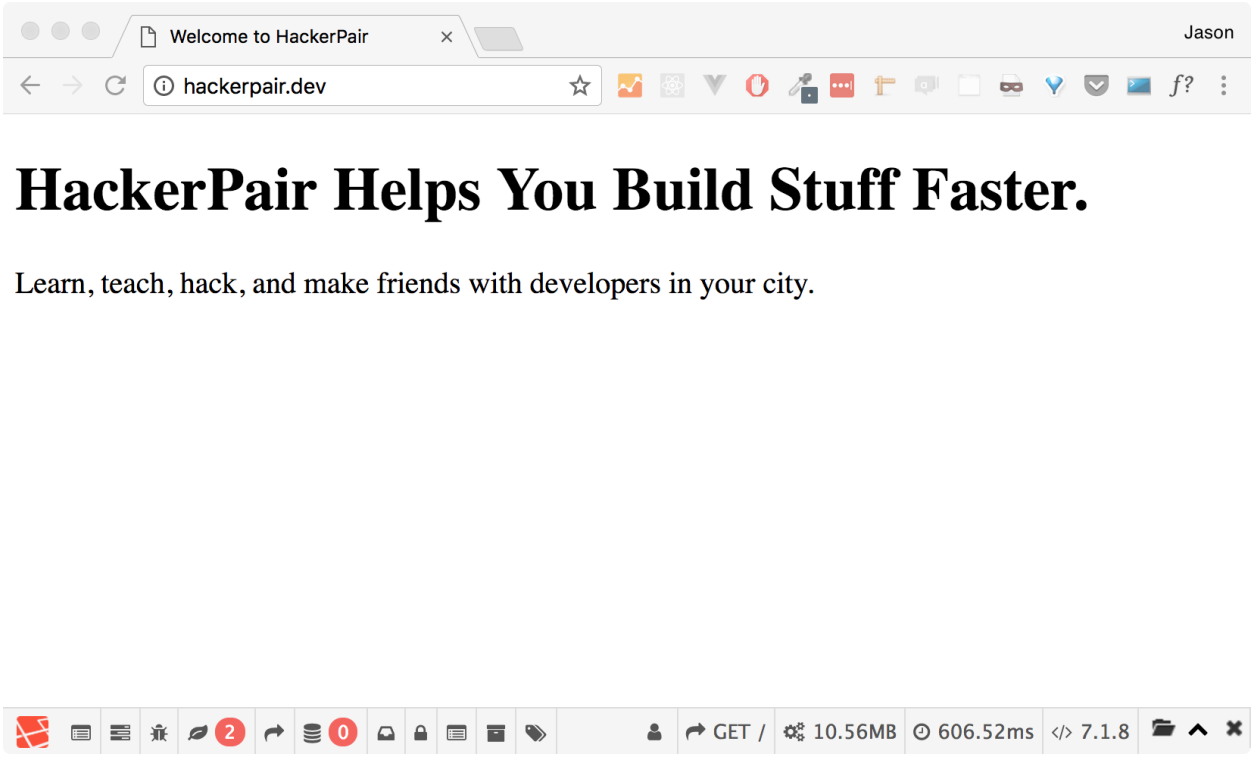
```
<!doctype html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Welcome to HackerPair</title>
</head>
<body>

  @yield('content')

</body>
</html>
```

The `@yield` directive identifies the name of the *section* that should be embedded into the template. This is best illustrated with an example. After saving the changes to `app.blade.php`, open `welcome.blade.php` and modify its contents to look like this:

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

@section('content')

    <h1>HackerPair Helps You Build Stuff Faster.</h1>

    <p>
      Learn, teach, hack, and make friends with developers
      in your city.
    </p>

@endsection
```

The `@extends` directive tells Laravel which layout should be used. Note how dot notation is used to represent the path, so for instance `layouts.app` translates to `layouts/app`. You specify the layout because it's possible your application will employ multiple layouts, for instance one sporting a sidebar and another without.

After saving the changes reload the home page and you should see the output presented in the following screenshot.



Wrapping the home page in a layout

Defining Multiple Layout Sections

A layout can identify multiple sections which can then be overridden or enhanced within each view. For instance many web applications employ a main content area and a sidebar. In addition to the usual header and footer the layout might include some globally available sidebar elements, but you probably want the flexibility of appending view-specific sidebar content. This can be done using multiple `@section` directives in conjunction with `@show` and `@parent`. For reasons of space I'll just include the example layout's `<body>` contents:

```
<div>
    @yield('content')
</div>

<div>
    @section('advertisement')
        <p>
            Score some HackerPair swag in our store!
        </p>
    @show
</div>
```

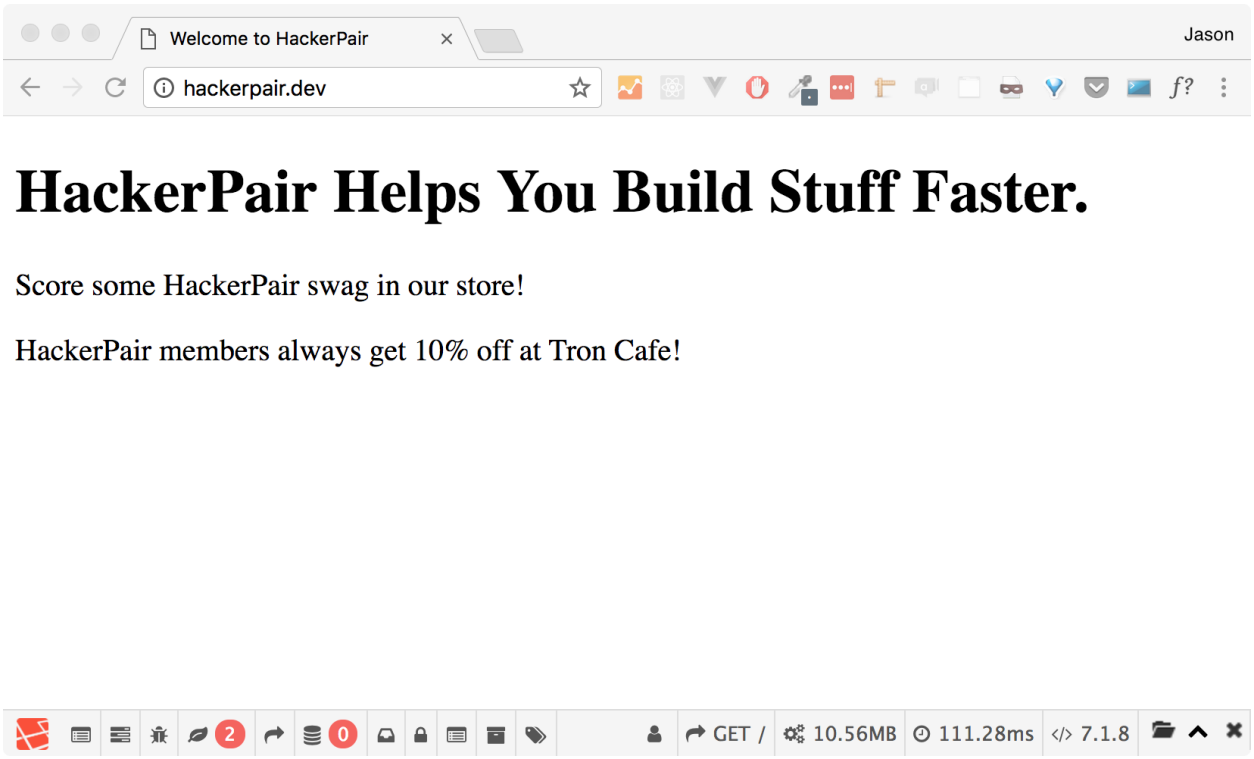
The `@show` directive as a shortcut for closing the section and then immediately yielding it. It's equivalent to this:

```
@endsection
@yield('advertisement')
```

The view can then also reference `@section('advertisement')`, additionally referencing the `@parent` directive which will cause anything found in the view's sidebar section to be *appended* to anything found in the layout's sidebar section:

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

@section('content')
  <h1>HackerPair Helps You Build Stuff Faster.</h1>
@endsection

@section('advertisement')
  @parent
    <p>
      HackerPair members always get 10% off at Tron Cafe!
    </p>
@endsection
```

Once this view is rendered, the `advertisement` section would look like this:



Managing Multiple Layout sections

If you would rather replace (rather than append to) the parent section, just eliminate the `@parent` directive reference.

Taking Advantage of View Partials

Suppose you wanted to include a widget within several different areas of the application. This bit of markup is fairly complicated, such as a detailed table row, and you assume it will be subject to considerable evolution in the coming weeks. Rather than redundantly embed this code within multiple locations, you can manage it within a separate file (known as a *view partial*) and then include it within the views as desired. For instance, if you wanted to manage a table row as a view partial, create a directory named `partials` inside the `resources/views` directory. Inside this newly created directory create a file named for instance `_row.blade.php`. Add the table row markup to the file:

```
<tr>
  <td>
  {{ $event }}
  </td>
</tr>
```

Notice how I'm using a view variable (`$event`) in the partial. When importing the partial into your view you can optionally pass a variable into the view like so:

```
<table>
@foreach ($events as $event)

  @include('partials._row', ['event' => $event])

@endforeach
</table>
```

The use of a specially designated directory for managing your partials (`resources/views/partials` in the above example) is purely optional, as is prefixing partial names with an underscore. I do so solely for organizational purposes; you're free to manage the partials in any other manner you see fit.

Integrating Images, CSS and JavaScript

Your project images, CSS and JavaScript should be placed in the project's `public` directory. While you could throw everything into `public` and just reference the appropriate path within your view files, you'll see Laravel has already created `css` and `js` directories for you. I prefer to additionally create an `img` directory for managing images.

Remember from chapter 1 the `public` directory is the application's designated document root, meaning when you reference for instance an image, you won't prefix the path with `/public`. Instead, presuming you've followed my lead and created an `img` directory inside `public`, you'd embed an image into one of your views or layouts like this:

```
<img src="/img/logo.png" />
```

Recall from earlier in the chapter how many Laravel developers prefer to programmatically generate hyperlinks? The same convention is often followed for referencing application assets such as images, CSS and JavaScript files. While the native Laravel framework offers a few helpers for referencing assets stored in the `public` directory, I don't find them to be particularly convenient because they require you to intertwine the helpers with raw HTML, like this:

```
<img src="{{ asset('img/logo.png') }}" />
```

I think this is kind of messy, and so instead regularly take advantage of a few helpers available via the Laravel Collective HTML package[43]. For instance, the following three statements are identical:

```
<!-- Standard HTML markup -->
<img src="/img/logo.png" alt="HackerPair logo" />

<!-- Laravel's asset() helper -->
<img src="{{ asset('img/logo.png') }}" />

<!-- Using the LaravelCollective/html package -->
{!! HTML::image('img/logo.png', 'HackerPair logo') !!}
```

Similar HTML component helpers are available for CSS and JavaScript. Again, you're free to use standard HTML tags or can use the facade. The following two sets of statements are identical:

```
<!-- Standard HTML markup -->
<link rel="stylesheet" href="/css/app.min.css">
<script src="/js/jquery/3.2.1/jquery.min.js"></script>
<script src="/js/app.js"></script>

<!-- Using the LaravelCollective/html package -->
{!! HTML::style('css/app.min.css') !!}
{!! HTML::script('js/jquery.min.js') !!}
{!! HTML::script('js/app.js') !!}
```

If you want to take advantage of these HTML helpers (and I highly recommend you do), you'll need to install the `LaravelCollective/HTML` package. This was previously part of the native Laravel distribution, but has been moved into a separate package as of version 5. Fortunately, installing the package is easy. First, use Composer to install the package:

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

Because this package takes advantage of Laravel 5.5's autodiscovery feature, you'll only additionally need to append the `HTML` alias to your `config/app.php` `aliases` array:

[43] https://github.com/LaravelCollective/html

```
'aliases' => [
    'App' => Illuminate\Support\Facades\App::class,
    ...
    'HTML' => Collective\Html\HtmlFacade::class
],
```

Be sure to check out the GitHub README[44] of the Laravel documentation for a list of available helpers. I suggest taking the time to do so, because I find LaravelCollective/HTML to be one of Laravel's indispensable packages, particularly so for designing web forms, a topic we'll discuss in great detail in chapter 5.

Introducing Laravel Mix

Writing code is but one of many tasks the modern developer has to juggle when working on even the simplest of projects. You'll also typically want to compress images, minify CSS and JavaScript files, hide debugging statements from the production environment, run unit tests, and perform countless other mundane yet important duties. Keeping track of these responsibilities, let alone ensuring you remember to complete them, can be a pretty tall order.

The Laravel developers hope to reduce some of the time and hassle associated with these sort of tasks by providing an API called Laravel Mix[45]. In this section you'll learn how to create and execute Mix tasks in order to more effectively manage your project.

Installing Laravel Mix

Mix relies upon a number of Node.js[46] dependencies, meaning you'll need to install Node.js and the NPM package manager. No matter your operating system this is easily done by downloading one of the installers via the Node.js website[47]. If you'd prefer to build Node from source you can download the source code via this link. Mac users can install Node via Homebrew, while Linux users can install Node via their distribution's package manager.

Once installed you can confirm Node and NPM are accessible via the command-line by executing the following commands:

[44] https://github.com/LaravelCollective/html
[45] https://github.com/JeffreyWay/laravel-mix
[46] http://nodejs.org
[47] http://nodejs.org/download/

```
$ node -v
v8.4.0

$ npm -v
5.5.1
```

Node users have access to a great number of third-party libraries known as Node Packaged Modules (npm). You can install these modules via the aptly-named `npm` utility. We'll use `npm` to install a set of npm packages intended for use with a typical modern Laravel application:

```
$ npm install
```

So what happens when you run this command? Take a look at the `package.json` file found inside the project's root directory. As of Laravel 5.5 the file looks like this:

{ "private": true, "scripts": { "dev": "npm run development", "development": "cross-env NODE_ENV=development node_modules/...", "watch": "cross-env NODE_ENV=development node_modules/...", "watch-poll": "npm run watch – –watch-poll", "hot": "cross-env NODE_ENV=development node_modules/...", "prod": "npm run production", "production": "cross-env NODE_ENV=production node_modules/..." }, "devDependencies": { "axios": "0.17", "bootstrap_sass": "3.3.7", "cross-env": "5.1", "jquery": "3.2", "laravel-mix": "1.0", "lodash": "4.17.4", "vue": "^2.1.10" } }

The `devDependencies` section defines a number of packages which will be installed when the `npm install` command is executed. These include jQuery, Vue (see chapter 13), bootstrap-sass, and Laravel Mix, among others. The `scripts` section contains a list of command shortcuts you'll soon be using to build CSS and for other tasks.

Once the `install` command has completed, you'll find a new directory named `node_modules` has been created within your project's root directory, and within in it you'll find the package directories. This directory is used by Node.js to house the various packages installed per the `package.json` specifications, so you definitely do not want to delete nor modify it.

Running Your First Mix Task

Your Laravel project includes a default `webpack.mix.js` which defines your Mix tasks. Inside this file you'll find an example task:

```
mix.js('resources/assets/js/app.js', 'public/js')
   .sass('resources/assets/sass/app.scss', 'public/css');
```

This task actually chains together two separate tasks; The first, `mix.js`, is used to compile ECMAScript 2015 (also referred to as ES6) code into browser-supported JavaScript. The second task,

`mix.sass`, compiles Sass[48] files into CSS which can then be imported into your project layout. Let's focus on the former task first.

The `mix.sass` task compiles a file named `app.scss` which resides in `resources/assets/sass`. The default `app.scss` file contains the following CSS import statements:

```
// Fonts
@import url("https://fonts.googleapis.com/css?family=Raleway:300,400,600");

// Variables
@import "variables";

// Bootstrap
@import "~bootstrap-sass/assets/stylesheets/bootstrap";
```

When the `webpack.mix.js` file is executed (I'll show you how to do this in a moment), these import statements will result in the Google-hosted Raleway font, the `_variables.scss` file (found in `resources/assets/sass`), and the Bootstrap 3 framework CSS being integrated into a single CSS file and saved to `/public/css/app.css`. You can then reference this compiled CSS file within your project layout file's header like so:

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

Or if you're using the LaravelCollective/html package, you can reference it like so:

```
{!! HTML::style('css/app.css') !!}
```

Further, because `app.scss` is Sass-based, you can take advantage of its powerful CSS extensions to more effectively manage your CSS declarations. For instance if you open `_variables.scss` you'll see several CSS variables (that's right, CSS *variables*) are defined:

```
// Body
$body-bg: #f5f8fa;

// Borders
$laravel-border-color: darken($body-bg, 10%);
$list-group-border: $laravel-border-color;
$navbar-default-border: $laravel-border-color;
$panel-default-border: $laravel-border-color;
$panel-inner-border: $laravel-border-color;
```

[48] http://sass-lang.com/

```
10
11  ...
12
13  // Panels
14  $panel-default-heading-bg: #fff;
```

Because the imported Bootstrap CSS is Sass-enabled, these variables will be used within the Bootstrap stylesheet and compiled to `app.css`. You can peruse a large list of available variables within the bootstrap-sass GitHub repository (https://github.com/twbs/bootstrap-sass/blob/master/assets/stylesheets/bootstrap/_variables.scss).

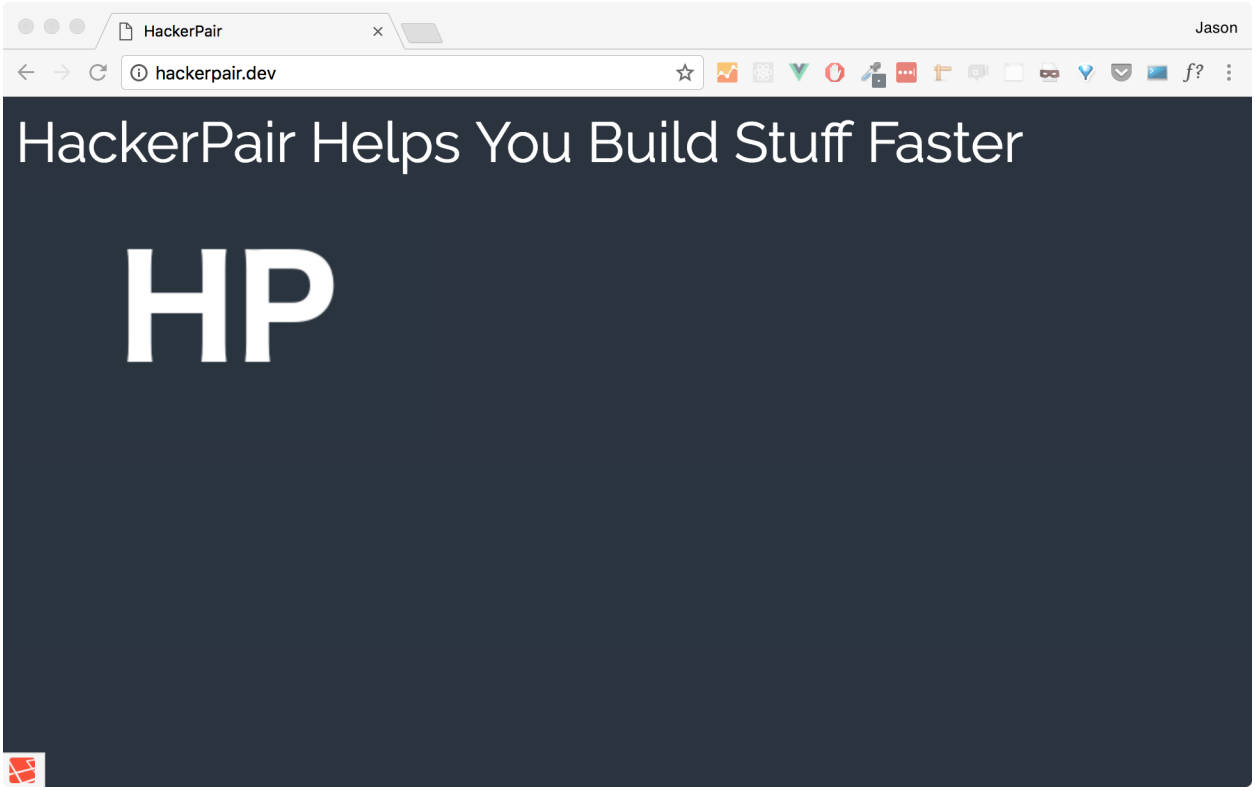
Just for kicks let's change the `$body-bg` variable from `#f5f8fa` to a dark blue (`#2A343E`). Additionally, change the `$text-color` variable from `#636b6f` to white `#ffffff`. After saving the changes execute the `webpack.mix.js` file by running `npm run dev` within your project root directory:

```
1  $ npm run dev
2  ...
3  DONE  Compiled successfully in 3402ms
4  ...
```

Once complete, you'll find a compiled CSS file named `app.css` inside your project's `public/css` directory. In order to use the styles defined in the `app.css` file you'll need to reference it within the head of your layout file (`resources/views/layouts/app.blade.php`). If you generated the `app.blade.php` file as described earlier in the chapter, this has already been done for you using Laravel's asset import syntax:

```
1  <link href="{{ asset('css/app.css') }}" rel="stylesheet">
```

Return to the browser and reload the home page. As of this point in the chapter my home page looks like this.



A slightly improved home page

Be sure to check out the Sass guide[49] for a comprehensive summary of what's possible with this powerful CSS extension syntax.

Watching for Changes

Because you'll presumably be making regular tweaks to your CSS and JavaScript, consider using the `watch` command to automatically execute `webpack.mix.js` anytime your assets change:

```
$ npm run watch
```

This process will continue to run, so you'll want to execute it in a dedicated tab or in the background. Once running, each time the target files associated with the Mix tasks are changed, the `webpack.mix.js` file will automatically be executed.

Integrating Bootstrap 4

At the time of this writing, Bootstrap 3 had been Laravel's "preset" CSS framework for quite some time, meaning (as you just learned) Bootstrap could be integrated into a Laravel application with

[49] http://sass-lang.com/guide

minimal effort. Yet Bootstrap 4 has been under active development for more than two years, and the third beta release has recently been released. Indeed, the new version of Bootstrap is creeping so close to a production release that on November 9, 2017 Taylor Otwell tweeted[50] "I figure it will be time to have a Bootstrap 4 preset in the core for Laravel 5.6".

If you don't want to wait for Laravel 5.6 or the Bootstrap production release (as I've done with HackerPair), then you can take advantage of a third-party preset solution now. It's called the "Laravel 5.5 Frontend Preset for Bootstrap 4", and is available here[51]

```
$ composer require laravelnews/laravel-twbs4
```

Next you'll update `package.json`, a variety of views (including `resources/views/layouts/app.blade.php` and `resources/views/welcome.blade.php`), and various `assets/sass` files to take advantage of Bootstrap 4. Read that last sentence twice, or maybe even three times. If you've spent any time at all tweaking any of these files, then you'll want to back up these files (or better, commit them to your Git repository). After you've taken care to make these files recoverable if necessary, run the following command:

```
$ php artisan preset bootstrap4-auth
Bootstrap (v4) scaffolding with auth views installed successfully.
Please run "npm install && npm run dev" to compile your fresh scaffolding.
```

In addition to changing the aforementioned files, a new directory named `auth` has been created in your `resources/views/directory`. This directory contains various authentication- and account-related files. The reason why this seemingly incongruously was suddenly created is because the Bootstrap 4 preset wants to make sure the authentication view markup is updated to use Bootstrap 4-specific syntax. If you don't plan on integrating authentication- or account-related features into your application, you can instead execute:

```
$ php artisan preset bootstrap4
```

Finally, you'll need to install the Bootstrap 4-specific version of the bootstrap-sass package. This is done by running `npm install` anew, because when you ran the previous `preset` command it updated your project's `package.json` file, replacing the following line:

```
"bootstrap-sass": "^3.3.7",
```

with:

[50] https://twitter.com/taylorotwell/status/928702823319195648

[51] composerrequirelaravelnews/laravel-twbs4

```
"bootstrap": "^4.0.0-beta.2",
```

If you're wondering why `bootstrap-sass` has been removed altogether, it's because Bootstrap 4 uses Sass by default, meaning there's no need for a custom npm package which melds the two together.

Finally, run `npm run dev` to rebuild your project assets:

```
$ npm run dev
```

Once complete, your project can take advantage of Bootstrap 4's many exciting new features!

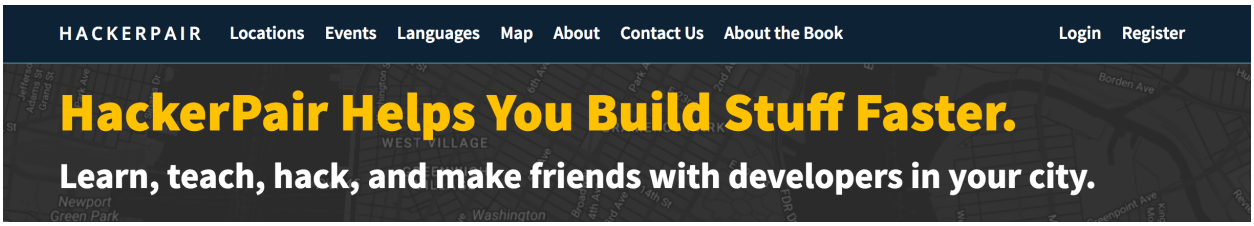
Project Exercise: Build the HackerPair Skeleton

By this point in the book you've learned enough to create a well structured skeleton application which will form the foundation for future chapters. So let's put what you've learned into action by generating the initial set of controllers, views, and routes used to power the application.

Keep in mind this is just a learning exercise! You're certainly not required to sort out all of these details at the onset of a Laravel project. Also, don't worry if you haven't yet integrated Bootstrap or another CSS framework. Nothing is going to prevent you from doing this later and subsequently marking up each view file with the necessary HTML and CSS as the application progresses. Finally, realize making mistakes is just part of the learning process. I've lost count of the number of controllers, views, and other files I've deleted over the past few years. Making mistakes is no big deal at this point, so just keep learning from those mistakes and have fun!

Step #1. Create the Project Controllers

If you head over to `http://hackerpair.com` you'll notice the top navigation bar (see below screenshot) includes links to a variety of site sections.



The HackerPair Navigation Bar

Based on the link titles, one can reasonably conclude that the `Locations`, `Events`, `Languages`, and `Map` sections will be database-driven and therefore we'll want to serve those views by way of a controller. For the time being the `About`, `Contact Us`, `About the Book` sections could get by as static HTML files. In chapter 8 we'll convert the contact page from a static file to a form-driven solution which sends the user inquiry to support via e-mail. The `Login` and `Register` sections are handled largely by Laravel and aren't anything we need to be concerned about until chapter 9. So let's create the controllers:

```
$ php artisan make:controller EventsController
$ php artisan make:controller LanguagesController
$ php artisan make:controller LocationsController
$ php artisan make:controller MapsController
```

You may have already created the `Events` controller when following along with earlier chapter exercises. If so, just move on to creating the `Languages` controller.

Open each controller and add a lone `index` method:

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

You'll need to update the view path accordingly for each controller. For instance in the above example `events.index` corresponds to the `resources/views/events/index.blade.php` view (which we'll create in a moment). In the `Languages` controller you'll point to `languages.index`, and so on.

Step #2. Create the Views

Next let's create the views associated with each controller's `index` method. Presuming you want to manage each controller's set of views in a separate folder, navigate to the `resources/views` directory and create new directories named `events` (if you haven't already), `languages`, `locations`, and `maps`. Then create a file named `index.blade.php` and add the following contents to it:

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

@section('content')

<h1>Events</h1>

@endsection
```

Save this file to the `resources/views/events` directory (again if you haven't already done so), and then copy the file into the `languages`, `locations`, and `maps` directories, changing the `h1` header text accordingly as you go along.

Next, we need to create directories for the view-driven sections (`About`, `About the Book`, and `Contact Us`). Inside `resources/views` create a directory named `about`, and inside it create two new files named `index.blade.php` and `book.blade.php`. Copy the above view contents into those files and once again modify the `h1` contents accordingly. Lastly, create another directory named `contact` and copy and modify `index.blade.php` one final time.

Step #3. Wire Up the Routes

The controllers and views are in place, but the application doesn't yet know how to connect them to user requests! That's the job of the `routes/web.php` file, so open it up and add the necessary `Route::get` and `Route::view` statements. As of this point in the book, my application `web.php` file looks like this:

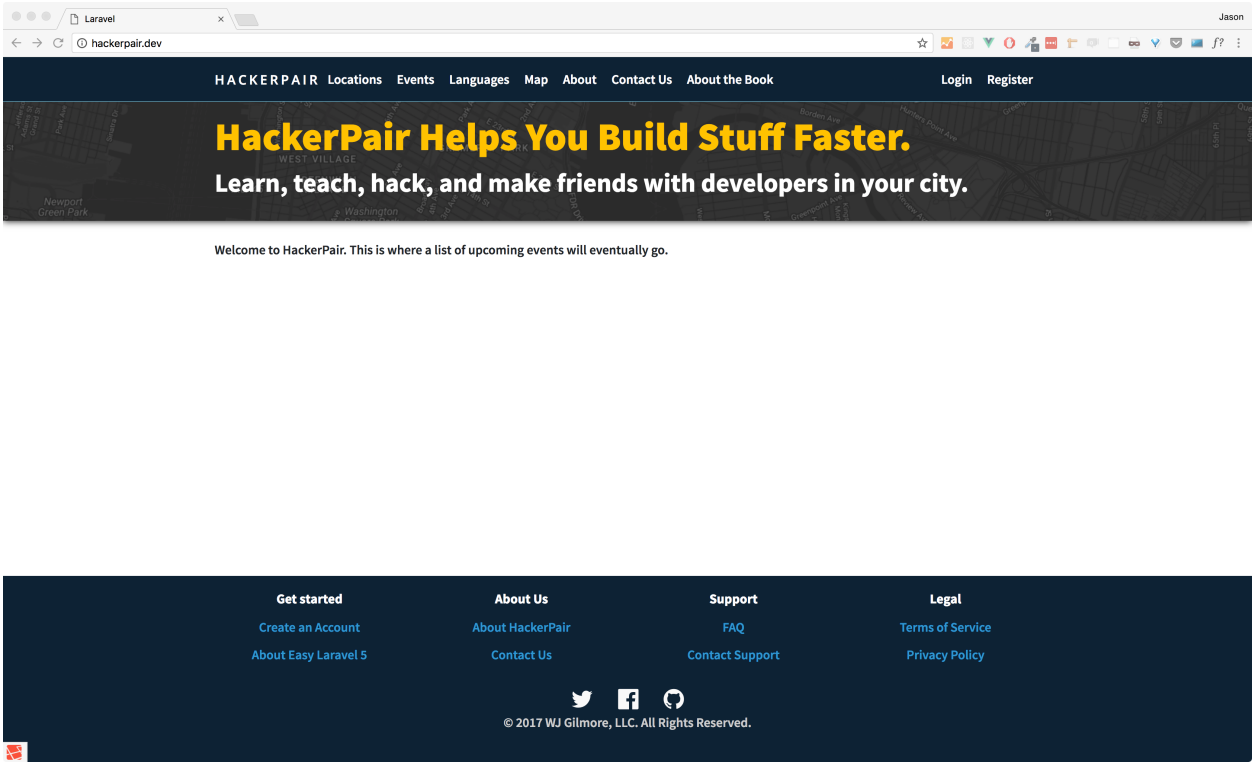
```
Route::get('/', 'WelcomeController@index');

Route::view('about', 'about.index')->name('about.index');
Route::view('about/book', 'about.book')->name('about.book');
Route::view('about/faq', 'about.faq')->name('about.faq');
Route::view('about/privacy', 'about.privacy')->name('about.privacy');
Route::view('about/tos', 'about.tos')->name('about.tos');

Route::view('contact', 'contact.index')->name('contact.index');

Route::get('events', 'EventsController@index')->name('events.index');
Route::get('events/{id}', 'EventsController@show')->name('events.show');

Route::get('languages', 'LanguagesController@index')->name('languages.index');

Route::get('locations', 'LocationsController@index')->name('locations.index');

Route::get('map', 'MapsController@index')->name('maps.index');

Auth::routes();
```

This example includes a few extra routes which point to the site's privacy policy and terms of service. You certainly don't need to include those in your practice project; I'm just including them for sake of reference.

Step #4. Manually Test the Routes

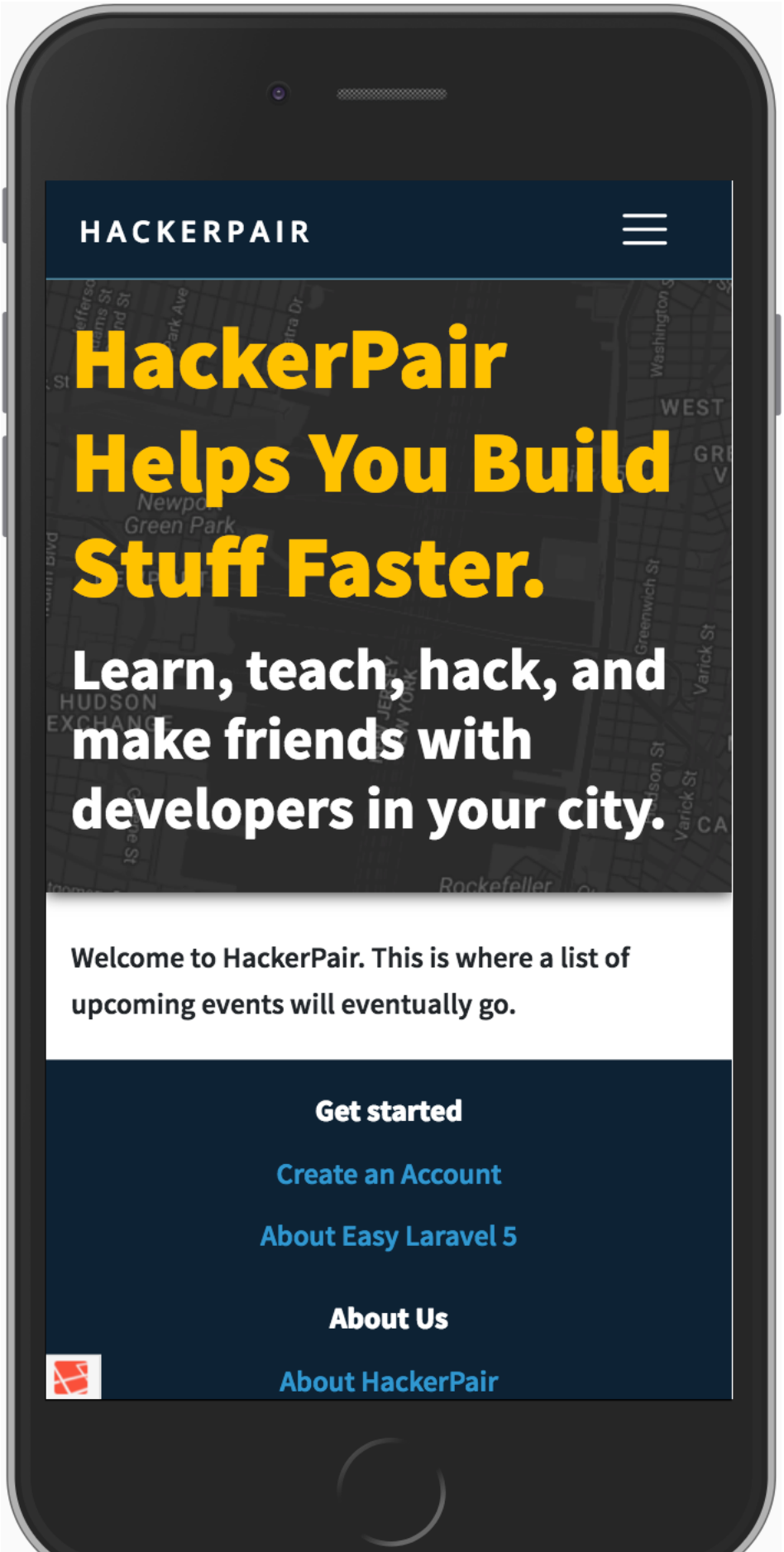
With the initial pieces in place, spend a few minutes just clicking around to make sure your routes are properly wired up. Don't be afraid to make a few early content-related tweaks to the views, incorporate some CSS markup if you've already integrated a CSS framework, or change some route names.

After completing these steps mysql and integrating Bootstrap 4, and making an agonizing series of UI tweaks because I'm horrible at that sort of thing, here's what my HackerPair project skeleton looks like in desktop view:



The HackerPair Skeleton Desktop View

And here's what it looks like in mobile view:


HACKERPAIR
HackerPair Helps You Build Stuff Faster.
Learn, teach, hack, and make friends with developers in your city.
Welcome to HackerPair. This is where a list of upcoming events will eventually go.
Get started
Create an Account
About Easy Laravel 5
About Us
About HackerPair

Testing the Project Skeleton with Laravel Dusk

At multiple points throughout the course of learning web development you've undoubtedly spent time repeatedly clicking through a project's pages and filling out forms in order to ensure everything was working correctly. I know because I've been there and done that, many times. This is occasionally a fun and even motivational task because it gives you the opportunity to understand from the user's perspective just how much of the project has been completed (not to mention actually working), but it is by no means something you should be doing on a regular basis. Manual testing of this nature is an inefficient use of time, distracting, and ultimately not an effective way to continuously ensure your application is operational at any given point in time.

But somebody has confirm that your giant ball of code is operational, right? Shouldn't somebody, but rather *something*, namely Laravel's excellent automated testing features, bear this important responsibility. In the last chapter you learned about *unit tests*, which are intended to test discrete bits of code, and *feature tests*, which confirm the behavior of multiple code units working together. Even so, feature tests can only do so much, because they are largely limited to making a network call to a particular route (or set of routes), examining HTTP headers, and parsing the results to confirm among other things whether a particular string is present. They can't execute JavaScript, nor can they ensure a page renders properly within a specific browser such as Chrome or Firefox.

To account for this deficiency, there is one more testing approach you'll need to take into consideration to really round out your understanding of this topic: *integration testing*. Integration tests can be thought of as feature tests on steroids, because while they also involve ensuring multiple code units are working together, they do so by simulating a user interacting with the application via an actual web browser. Let that last sentence sink in. We're going to write automated tests which actually interact with a browser in order to confirm your Laravel project is behaving as it should. Because of this, you have the opportunity to test JavaScript behavior, ensure your pages are rendering exactly as they should within the browser, and any of the many other issues which may only be identified by "human" eyes. Read on to learn how!

Installing Laravel Dusk

Laravel Dusk is available as a standalone package. To install it, run the following command from your project's root directory:

```
1 $ composer require --dev laravel/dusk
```

Next you'll need to add the following code to the `app\Providers\AppServiceProvider.php` `register` method:

```
if ($this->app->environment('local', 'testing')) {
    $this->app->register(DuskServiceProvider::class);
}
```

Once that's done run Artisan's `dusk:install` command to complete installation:

```
$ php artisan dusk:install
Dusk scaffolding installed successfully.
```

With installation complete, you'll find a new directory in your project's `tests` directory named `Browser`, in addition to a new file inside `tests` named `DuskTestCase.php`:

- `Browser`: This directory is used to manage your Dusk tests, Dusk helpers, and various debugging devices. It contains four subdirectories, in addition to an example test named `ExampleTest.php`. I'll discuss the role of these subdirectories and dive into `ExampleTest.php` in a moment.
- `DuskTestCase.php`: Just as the test classes we experimented with in chapter 1 inherit from `TestCase.php`, Dusk test classes inherit from `DuskTestCase.php`. This class is responsible for starting and configuring the Chrome driver. We'll make a temporary tweak to this class in a bit so you can actually see what happens when Dusk is running the tests.

Inside the `Browser` directory you'll find the following directories and files:

- `Components`: This directory contains reusable sets of steps taken to interact with a specific component, such as the HackerPair event search filter. You could create a Dusk component which interacts with the filter language, date range, and location range, and then reuse this component throughout your tests, passing along the desired values for each filter setting. We'll build custom Dusk components in later chapters.
- `Pages`: This directory contains reusable sets of steps taken to interact with a specific page which you can incorporate into other tests, saving you the hassle of recreating the code inside each new test. For instance several of HackerPair's Dusk tests involve creating new events; because doing so involves filling out the event creation form, we'll bundle those steps into a Dusk page and then reuse it whenever necessary within other tests. In later chapters I'll show you how this is done.
- `screenshots`: Dusk offers an amazing feature in which when a test fails, it will take a screenshot of the screen and save it to this folder, allowing you to see the problem from the perspective of a user! I'll demonstrate this feature in just a moment.
- `ExampleTest.php`: Inside this file you'll find an example Dusk test. We'll have a look at this test next.

Running the Example Test

Dusk includes an example test which navigates to the application home page and determines whether the string `Laravel` is found in the page content. It's found in `tests/Browser/ExampleTest.php`, and it looks like this:

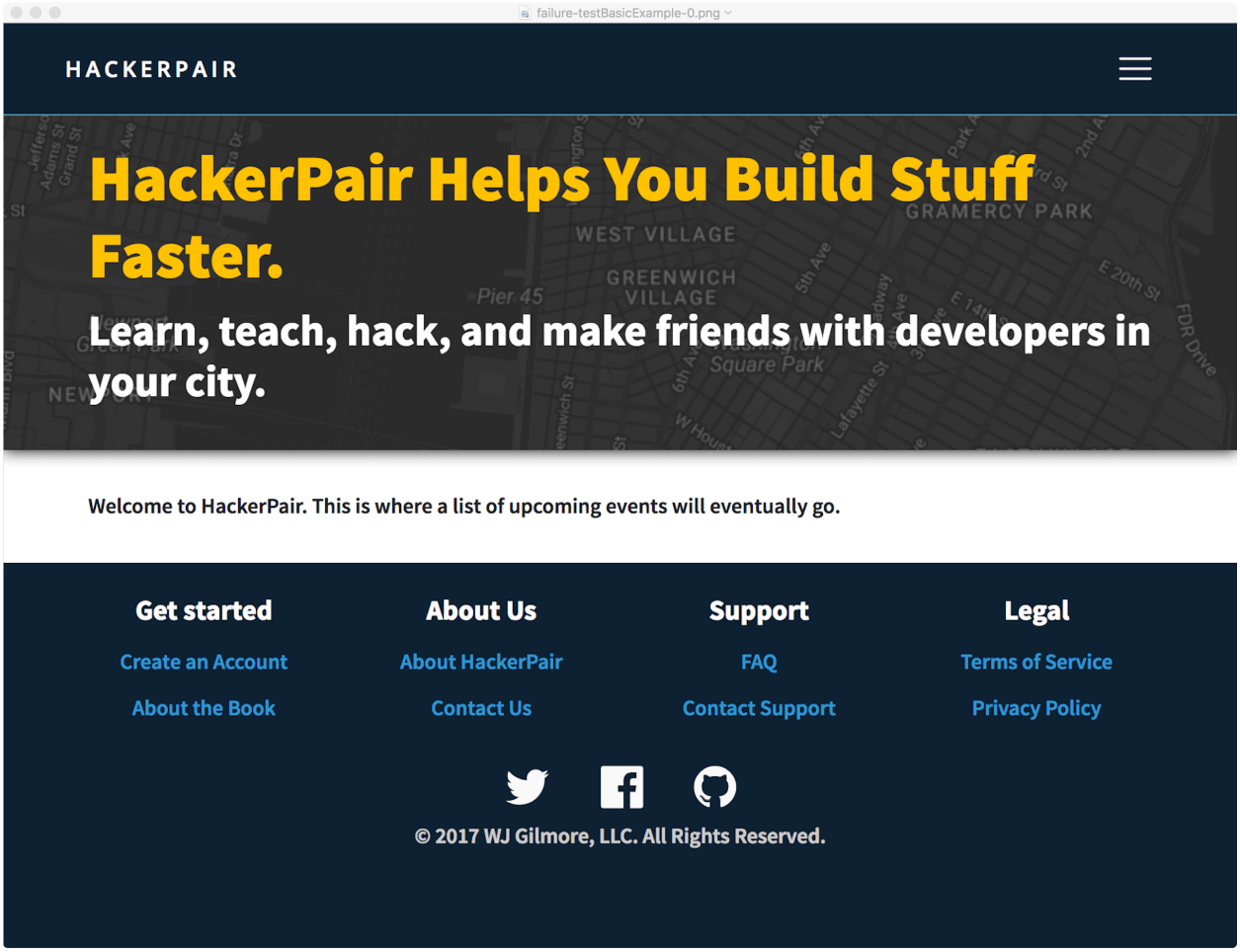
```
<?php

namespace Tests\Browser;

use Tests\DuskTestCase;
use Laravel\Dusk\Browser;
use Illuminate\Foundation\Testing\DatabaseMigrations;

class ExampleTest extends DuskTestCase
{
    /**
     * A basic browser test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->browse(function (Browser $browser) {
            $browser->visit('/')
                    ->assertSee('Laravel');
        });
    }
}
```

As you can see, it looks pretty similar to the test code introduced in chapter 1. A primary difference though is you won't run Dusk tests using `vendor/php/phpunit`. Instead, you'll use Artisan's newly added `dusk` command:

```
$ php artisan dusk tests/Browser/ExampleTest.php
PHPUnit 6.4.3 by Sebastian Bergmann and contributors.

F                                   1 / 1 (100%)

Time: 1.92 seconds, Memory: 12.00MB

There was 1 failure:

1) Tests\Browser\ExampleTest::testBasicExample
Did not see expected text [Laravel] within element [body].
Failed asserting that false is true.

/Users/wjgilmore/Code/valet/hackerpair/vendor/laravel/dusk/src/...
/Users/wjgilmore/Code/valet/hackerpair/vendor/laravel/dusk/src/...
/Users/wjgilmore/Code/valet/hackerpair/tests/Browser/ExampleTest.php:20
/Users/wjgilmore/Code/valet/hackerpair/vendor/laravel/dusk/src/...
/Users/wjgilmore/Code/valet/hackerpair/tests/Browser/ExampleTest.php:21

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

My test failed because I've already made significant changes to the project home page. If you haven't yet updated the home page then this test will pass in the same fashion as those executed in chapter 1. I *want* you to make it fail, so if it passes open up your root view, change the term `Laravel` to anything but, and then run the test again to ensure it fails.

Because it failed, Dusk took a screenshot of the page which caused the failed test. Here's what my screenshot looks like:



A Dusk Screenshot

This particular screenshot isn't useful since no error-related information is presented on the page. But it does give you the opportunity to scan the page contents in order to see why the test logic conflicts with reality. Later in the section you'll see some examples in which the screenshot offers obvious feedback regarding the nature of the error.

Also, you will notice some occasional oddities associated with these screenshots. For instance, in the above example you may have noticed the navigation items are mysteriously absent. I'm not entirely sure why behavior randomly crops up, and just presume it's due to some sort of bug associated with the screenshot-related feature. Regardless, this ability provides a huge advantage to developers so just roll with it!

Viewing the Test

When Dusk tests run, the output mimics that seen when running the PHPUnit-driven tests presented in the last chapter. But Dusk tests actually involve a browser, which raises the question: where is the browser? The browser is running in what's referred to as *headless mode*, meaning it is indeed running

despite the browser user interface remaining hidden. You can disable headless mode and watch the tests actually interact with the browser by opening `tests/DuskTestCase.php` and commenting out the `--headless` line in the `driver` method. To be clear, your `DuskTestCase.php` class' `driver` method should look like this once done:

```
protected function driver()
{
    $options = (new ChromeOptions)->addArguments([
        '--disable-gpu',
        //'--headless'
    ]);

    return RemoteWebDriver::create(
        'http://localhost:9515', DesiredCapabilities::chrome()->setCapability(
            ChromeOptions::CAPABILITY, $options
        )
    );
}
```

Save these changes and run `php artisan dusk` again. This time you'll see the Chrome browser briefly open and display the project home page. If you're paying really close attention you'll notice a warning just below the address bar that states "Chrome is being controlled by automated test software". This will all occur pretty quickly since we're currently only dealing with a single test, but as your test suite grows in size it can be occasionally interesting to disable headless mode and watch the browser open and close as each test executes. Most of the time you'll want to leave headless mode enabled because the tests run dramatically faster in this configuration.

Creating Your First Dusk Test

Let's create a simple Dusk test to ensure the links located in the HackerPair navigation bar work as expected. While not a particularly revealing test in the sense it might uncover some deeply hidden bug, it will give you the opportunity to become familiar with some pretty powerful testing methods. Begin by creating the test

```
$ php artisan dusk:make NavigationBarTest
Test created successfully.
```

Open the newly created file (`tests/Browser/NavigationBarTest.php`) and modify the `testExample` method, changing both its name and contents to look like this:

```
public function testLocationLinkTakesUserToLocationsIndex()
{
    $this->browse(function (Browser $browser) {
        $browser->visit('/')
                ->clickLink('Locations')
                ->assertPathIs('/locations');
    });
}
```

When invoked, this test will navigate to the application home page, click a hyperlink with the title `Locations`, and confirm the destination path is `/locations`. Now run just this test by passing along the `NavigationBarTest` path to the `dusk` command:

```
$ php artisan dusk tests/Browser/NavigationBarTest.php
PHPUnit 6.4.3 by Sebastian Bergmann and contributors.

.                                                   1 / 1 (100%)

Time: 2.56 seconds, Memory: 12.00MB

OK (1 test, 1 assertion)
```

Despite its' simple nature, I'd imagine this test already has your mind racing regarding the possibilities. No longer will you lose sleep wondering whether links are working! Just add another test to your suite and let it run with the others. And believe me, this only scratches the surface in terms of what we're going to do with Dusk in the chapters to come. In the meantime, try writing a few more tests to confirm your other navigation bar links are operational!

Incidentally, while I always recommend assigning a descriptive name to the test, admittedly the test names can quickly become difficult to read. You can alternatively use underscores to separate words in a test name rather than camel casing. For instance this is a perfectly acceptable alternative naming convention for the above test:

```
public function test_location_link_takes_user_to_locations_index()
{
    ...
}
```

Conclusion

You'll undoubtedly spend a lot of time and effort working on your project's layout, views, and CSS and so hopefully this chapter provided enough information and insight into managing these important assets. In the next chapter we'll dive deep into how your Laravel project's data is created, managed and retrieved. Onwards!

Chapter 3. Talking to the Database

Like most typical web applications, HackerPair manages a variety of data within a relational database. Events and users are the most obvious types of data found in the database, so you might be surprised to know the application relies on a total of almost a dozen different database tables. Further, although the HackerPair table schemas, data relations, and queries aren't particularly complicated, past experience dictates things can get very messy quickly if the database isn't properly maintained, or if little thought is put into how the application logic will integrate with the database.

Fortunately, the Laravel developers have put a tremendous amount of effort into ensuring both the ability to manage your database schema and integrate that database into your application is as seamless and convenient as possible. In the first of two chapters dedicated to these topics, you'll learn how to integrate a database into your application, create *models* which serve as the bridge between your application and the database, use *migrations* to manage your project's underlying database schema, and *seed* your database with useful test and helper data.

We'll spend the majority of the remaining chapter reviewing dozens of examples demonstrating how Laravel's *Eloquent* ORM can be used to retrieve, insert, update and delete data, before wrapping up with a look at Laravel's *Query Builder* interface (including a demonstration of how to execute raw SQL) and a section on testing your models.

This introductory material sets the stage for the next chapter, in which you'll learn about more advanced topics such as model relationships, collections, and scopes.

Configuring Your Project Database

In Chapter 1 I briefly touched upon Laravel's database support. To summarize, Laravel supports four databases, including MySQL, PostgreSQL, SQLite, and Microsoft SQL Server. The `config/database.php` file tells Laravel which database you'd like your application to use, what authentication credentials to use to connect to this database, as well as defines a few other data-related settings.

You won't typically modify the contents of this file directly; instead you'll define the desired database type, name, and connection credentials using environment variables (notably the `.env` file during development). Even so, it's beneficial to know what's going on in this file, and so I've pasted in a relevant snippet of `config/database.php` below (notably the MySQL-specific sections). Following the snippet I'll summarize the purpose of each setting:

```
return [

    'default' => env('DB_CONNECTION', 'mysql'),

    ...

    'connections' => [

    ...

    'mysql' => [
        'driver' => 'mysql',
        'host' => env('DB_HOST', '127.0.0.1'),
        'port' => env('DB_PORT', '3306'),
        'database' => env('DB_DATABASE', 'forge'),
        'username' => env('DB_USERNAME', 'forge'),
        'password' => env('DB_PASSWORD', ''),
        'unix_socket' => env('DB_SOCKET', ''),
        'charset' => 'utf8mb4',
        'collation' => 'utf8mb4_unicode_ci',
        'prefix' => '',
        'strict' => true,
        'engine' => null,
    ],

    ...

    ],

    'migrations' => 'migrations',

    ...

];
```

Let's review each setting:

- `default`: The `default` setting identifies the type of database used by your project. You'll set this using the `DB_CONNECTION` environment variable, assigning the variable a value of `mysql` (the default), `pgsql` (PostgreSQL), `sqlite` (SQLite), or `sqlsrv` (Microsoft SQL Server). Keep in mind none of these databases are installed when you generate a new Laravel application. You'll need to separately install and configure the desired database, or obtain credentials to access a remote database.

- `connections`: This array defines the database authentication credentials for each supported database. Laravel will only pay attention to the settings associated with the database identified by the `default` setting value (it is possible to configure Laravel to use multiple databases), therefore you can leave the unused database options untouched (or even entirely remove them from the configuration file). Several additional environment variables will then be used to supply the database's connection credentials. For instance, MySQL users will define the `DB_HOST`, `DB_PORT`, `DB_DATABASE`, `DB_USERNAME`, and `DB_PASSWORD`, or assume the default settings (`127.0.0.1`, `3306`, `forge`, `forge`, and no password, respectively).
- `migrations`: This setting defines the name of the table used for managing your project's migration status (more about this later in the chapter). This is by default set to `migrations`, but if by the wildest of circumstances you needed to change the table name to something else, you can do so here.

After identifying the desired database and defining the authorization credentials, don't forget to create the database because Laravel will not do it for you. If you're using Homestead then a database has already been created for you by way of the `Homestead.yaml` file, negating the need to go through these additional steps. With the database created, it's time to begin interacting with it!

Introducing the Eloquent ORM

Object-relational mappers (ORM) are without question the feature that led me to embrace web frameworks several years ago. Even if you've never heard of an ORM, anybody who has created a database-driven web site has undoubtedly encountered the problem this programming technique so gracefully resolves: *impedence mismatch*. Borrowed from the electrical engineering field, impedance mismatch[52] is the term used to describe the challenges associated with using an object-oriented language such as PHP or Ruby in conjunction with a relational database, because the programmer is faced with the task of somehow mapping the application objects to database tables. An ORM solves this problem by providing a convenient interface for converting application objects to database table records, and vice versa. Additionally, most ORMs offer a vast array of convenient features useful for querying, inserting, updating, and deleting records, managing table relationships, and dealing with other aspects of the data life cycle.

Creating Your First Model

Creating a model using Artisan is easy. Let's kick things off by creating the `Event` model, which the HackerPair application uses to manage user-created events. Generate a model by running the following command:

[52] http://en.wikipedia.org/wiki/Impedance_matching

```
$ php artisan make:model Event --migration
Model created successfully.
Created Migration: 2017_11_17_202416_create_events_table
```

Notice I also supplied the `--migration` option to the `make:model` command. This option tells Laravel to additionally create a table migration (we'll talk about migrations in a moment). The newly created model is found in `app/Event.php`. It looks like this:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Event extends Model
{
    //
}
```

It is vitally important to realize a Laravel model is just a PHP class that extends Laravel's `Model` class, thereby endowing the class with the features necessary to act as the bridge between your application and underlying database table. The class is currently empty, and we'll begin expanding its contents soon enough.

Like most web frameworks, Laravel expects the model name to be singular form (`Event`), and the underlying table names to be plural form (`events`). Further, when a model uses camel casing (e.g. `EventUser`), the corresponding table will use underscores to separate each capitalized word, and the trailing word pluralized (e.g. `event_users`).

A model is only useful when it's associated with an underlying database table. When we created the `Event` model we chose to additionally create a corresponding *migration*. This file contains the blueprint for creating the the model's associated table. Let's talk about the power of migrations next.

Introducing Migrations

With the model created, you'll typically create the corresponding database table, done through a fantastic Laravel feature known as *migrations*. Migrations offer a file-based approach to changing the structure of your database, allowing you to create and drop tables, add, update and delete columns, and add indexes, among other tasks. Further, you can easily revert, or *roll back*, any changes if a mistake has been made or you otherwise reconsider the decision. Finally, because each migration is stored in a text file, you can manage them within your project repository.

To demonstrate the power of migrations, let's have a look at the migration file that was created along with the `Property` model (presuming you supplied the `--migration` option). This migration file is named `2017_11_17_202416_create_events_table.php`, and it was placed in the `database/migrations` directory. Open up this file and you'll see the following contents:

```
<?php

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

class CreateEventsTable extends Migration
{

    public function up()
    {
        Schema::create('events', function (Blueprint $table) {
            $table->increments('id');
            $table->timestamps();
        });
    }

    public function down()
    {
        Schema::dropIfExists('events');
    }
}
```

Like the model, a Laravel migration is just a standard PHP class, except the class extends the `Migration` class. The `up()` and `down()` methods have special significance in regards to migrations, with the `up()` method defining what occurs when the migration is executed, and `down()` defining what occurs when the migration is reverted. Therefore the `down()` method should define what happens when you'd like to *undo* the changes occurring as a result of executing the `up()` method. Let's first discuss this migration's `up()` method:

```
public function up()
{
    Schema::create('events', function (Blueprint $table) {
        $table->increments('id');
        $table->timestamps();
    });
}
```

You'll regularly see the `Schema` class appear in migration files, because it is Laravel's solution for manipulating database tables in all manner of fashions. This example uses the `Schema::create` method to create a table named `properties`. An anonymous function (closure) passed along as the `Schema::create` method's second parameter defines the table columns:

- `$table->increments('id')`: The `increments` method indicates we want to create an automatically incrementing integer column that will additionally serve as the table's primary key. This is pretty standard procedure when creating relational database tables, so you're generally not going to remove nor modify this field.
- `$table->timestamps()`: The `timestamps` method tells Laravel to include `created_at` and `updated_at` timestamp columns, which will be automatically updated to reflect the current timestamp when the record is created and updated, respectively. Whether you choose to omit these fields is entirely up to you, although you'll generally want to keep them in place for bookkeeping and debugging purposes.

There are plenty of other methods useful for creating different column data types, setting data type attributes, and more. Be sure to check out the documentation[53] for a complete list, otherwise stay tuned as we'll cover various other methods in the examples to come.

We want each `events` table record to manage more than just a primary key and timestamps. Just to get things rolling, let's make sure each event is assigned a name, city, and description. Modify the `Schema::create()` body to include these fields:

```
Schema::create('events', function(Blueprint $table)
{
    $table->increments('id');
    $table->string('name');
    $table->string('city');
    $table->text('description')->nullable();
    $table->timestamps();
});
```

If you're not familiar with these column types I'll describe them next:

[53] https://laravel.com/docs/master/migrations#columns

- `$table->string('name')` and `$table->string('city')`: The `string` method indicates we want to create two variable character columns (commonly referred to as a `VARCHAR` by databases such as MySQL and PostgreSQL) named `name` and `city`. Remember, the `Schema` class is database-agnostic, and therefore leaves it to whatever supported Laravel database you happen to be using to determine the maximum string length, unless you use other means to constrain the limit.
- `$table->text('description')`: The `text` method indicates we want to create a text column (commonly referred to as `TEXT` by databases such as MySQL and PostgreSQL).

The example's `down()` method is much easier to understand because it consists of a single instruction: `Schema::dropIfExists('events')`. When executed it will remove, or *drop* the `events` table.

Now that you understand what comprises this migration file, let's execute it and create the `events` table:

```
$ php artisan migrate
Migration table created successfully.
Migrating: 2014_10_12_000000_create_users_table
Migrated:  2014_10_12_000000_create_users_table
Migrating: 2014_10_12_100000_create_password_resets_table
Migrated:  2014_10_12_100000_create_password_resets_table
Migrating: 2017_11_17_202416_create_events_table
Migrated:  2017_11_17_202416_create_events_table
```

Because this is the very first migration, Laravel will also create several other tables, including `migrations`, `password_resets`, and `users`. The `password_resets` and `users` tables are used to manage user-related data, and we'll talk about both in chapter 7. The `migrations` table is used to keep track of the database's structural evolution in terms of which migration files have been executed. For instance after running the first set of migrations the `migrations` table looks like this:

```
mysql> select * from migrations;
+------------------------------------------------+-------+
| migration                                      | batch |
+------------------------------------------------+-------+
| 2014_10_12_000000_create_users_table           |     1 |
| 2014_10_12_100000_create_password_resets_table |     1 |
| 2016_10_04_011840_create_events_table          |     1 |
+------------------------------------------------+-------+
```

Every time a migration is run, a record will be added to the `migrations` table identifying the migration file name, and the group, or *batch* in which the migration belongs. In other words, if you create for instance three migrations and then run `php artisan migrate`, those three migrations will

be placed in the same batch. If you later wanted to undo any of the changes found in any of those migrations, those three migrations would be treated as a group and rolled back together (you'll soon learn this default behavior can be overridden and specific migrations can be rolled back).

In examples where I display MySQL data as it appears in the database table, I'll use output from the native MySQL client. In practice, this is a pretty arcane way to interact with the database these days, although for purposes of book presentation this format works pretty well. For development work I've used the fantastic Sequel Pro (OSX only). A number of other popular GUI-based MySQL clients are available for other operating systems, including notably phpMyAdmin and MySQL Workbench.

Once complete, open your development database using your favorite database editor (I prefer to use the MySQL CLI), and confirm the `events` table has been created:

```
mysql> show tables;
+------------------------------+
| Tables_in_dev_hackerpair_com |
+------------------------------+
| events                       |
| migrations                   |
| password_resets              |
| users                        |
+------------------------------+
```

Indeed it has! Let's next check out the `events` table schema:

```
mysql> describe events;
+-------------+------------------+------+-----+---------+----------------+
| Field       | Type             | Null | Key | Default | Extra          |
+-------------+------------------+------+-----+---------+----------------+
| id          | int(10) unsigned | NO   | PRI | NULL    | auto_increment |
| name        | varchar(255)     | NO   |     | NULL    |                |
| city        | varchar(255)     | NO   |     | NULL    |                |
| description | text             | NO   |     | NULL    |                |
| created_at  | timestamp        | YES  |     | NULL    |                |
| updated_at  | timestamp        | YES  |     | NULL    |                |
+-------------+------------------+------+-----+---------+----------------+
```

Sure enough, an automatically incrementing integer column has been created, in addition to the `name`, `city`, `description`, `created_at` and `updated_at` columns.

Undoing Your Database Changes

Particularly in your project's early stages the database schema will undergo a great deal of flux. New ideas will pop up, old ideas will be reconsidered, and the schema will need to change accordingly. It is crucially important for you to realize tht during this development phase the database should be treated like a set of toy blocks; something that can be repeatedly built up and torn down without repercussion. Thanks to migrations, you have the luxury of doing this as many times as you see fit. For instance, suppose you meant to add a `venue` field to the `events` migration, but that migration has already been rolled into the database. No big deal! Just run the Artisan `migrate:rollback` command to undo the change:

```
$ php artisan migrate:rollback
Rolling back: 2017_11_17_202416_create_events_table
Rolled back:  2017_11_17_202416_create_events_table
Rolling back: 2014_10_12_100000_create_password_resets_table
Rolled back:  2014_10_12_100000_create_password_resets_table
Rolling back: 2014_10_12_000000_create_users_table
Rolled back:  2014_10_12_000000_create_users_table
```

Oh no! Because all of these table migrations resided in the same group, all of the corresponding table structures (and any data residing therein) have been removed from your development database. But ask yourself, is this really a big deal? Thanks to migrations, this act of building up and tearing down your database is as easy as executing a few Artisan commands. So go ahead and add the `venue` field to the `events` migration:

```
Schema::create('events', function(Blueprint $table)
{
    $table->increments('id');
    $table->string('name');
    $table->string('venue');
    $table->string('city');
    $table->text('description');
    $table->timestamps();
});
```

After saving your changes, run Artisan's `migrate` command to rebuild those data structures, including the modified `events` table.

Sometimes though, you really don't need to roll back all of the tables residing in a batch. For instance, because the `events` migration was the last to be created in the batch, you can easily roll back the changes for only the `events` table using the following command:

```
$ php artisan migrate:rollback --step=1
Rolling back: 2017_11_17_202416_create_events_table
Rolled back:  2017_11_17_202416_create_events_table
```

Notice I'm using `--step=1` here because we only want to remove the most recently created migration. If the migration you desire to modify resides in say the second-to-last migration in the recent batch, you could use `--step=2` to rollback the last two migrations.

After rolling back the changes check your database and you'll see both the `events` table has been removed and the relevant record in the `migrations` table. We'll actually want to use this table, so run `php artisan migrate` again before moving on.

Defining Column Modifiers

In many cases it simply isn't enough to just define a table's column names and associated data types. You'll additionally often want to further constrain the columns using modifiers. For instance, to set a column default you can use the `default` method:

```
$table->boolean('confirmed')->default(false);
```

To allow null values you can use the `nullable` method:

```
$table->string('comments')->nullable();
```

To set an integer column to unsigned, use the `unsigned` method:

```
$table->tinyInteger('age')->unsigned();
```

You can chain multiple modifiers to ensure even more sophisticated constraints:

```
$table->tinyInteger('age')->unsigned()->default(0);
```

You can review a complete list of supported column types and other options in the Laravel documentation[54].

Adding and Removing Columns

To add or remove a table column you'll generate a migration just as you did when creating a table in the previous section, the only difference being you'll use the `--table` option to identify the table you'd like to modify. For instance, some HackerPair users might wish to create events in advance but not immediately publish them to the site. We can give them the option of leaving the event disabled until an `enabled` column is set to true:

[54] http://laravel.com/docs/master/migrations

```
$ php artisan make:migration add_enabled_to_events_table --table=events
Created Migration: 2017_11_17_213116_add_enabled_to_events_table
```

Next open up the newly created migration file, creating the desired column in the `up` method, and making sure you drop the column in the `down` method:

```
public function up()
{
  Schema::table('events', function(Blueprint $table)
  {
      $table->boolean('enabled');
  });
}

public function down()
{
  Schema::table('events', function(Blueprint $table)
  {
      $table->dropColumn('enabled');
  });
}
```

Finally, run the migration to modify the `events` table:

```
$ php artisan migrate
Migrating: 2017_11_17_213116_add_enabled_to_events_table
Migrated:  2017_11_17_213116_add_enabled_to_events_table
```

Controlling Column Ordering

One of the beautiful aspects of an ORM such as Eloquent and migrations is the ability to essentially forget about matters such as column ordering. Even so, my penchant for needlessly obsessing over such details often leads to the desire to manage column ordering so as to ensure the primary and foreign keys are placed at the beginning of the table, and timestamps at the end of the table. If you would like to insert a column into a specific location within the table structure, use the `after` method:

```
$table->boolean('enabled')->after('name');
```

Other Useful Migration Commands

Sometimes you'll create several migrations and lose track of which ones were moved into the database. While you could visually confirm the changes in the database, Laravel offers a more convenient command-line solution. The following example uses the `migrate:status` command to review the status of the project database at some point during development:

```
$ php artisan migrate:status
+------+-----------------------------------------------+
| Ran? | Migration                                     |
+------+-----------------------------------------------+
| Y    | 2014_10_12_000000_create_users_table          |
| Y    | 2014_10_12_100000_create_password_resets_table |
| Y    | 2017_11_17_202416_create_events_table         |
| Y    | 2017_11_17_213116_add_enabled_to_events_table |
+------+-----------------------------------------------+
```

I also regularly use the `migrate:reset` and `migrate:refresh` commands to completely rollback and completely rebuild the migrations, respectively. The `migrate:reset` command is useful when you want to undo *all* migrations, since `migrate:rollback` will only undo one batch/step at a time. The `migrate:refresh` command is useful particularly when you make some adjustments to various migration files and would like to quickly tear down and rebuild all of the migrations.

Seeding the Database

Many beginning developers tend to spend an inordinate amount of time fretting over populating the development database with an appropriate amount of test data, often doing so by tediously completing and submitting a form. This is a waste of valuable time which can otherwise be spent improving and expanding the application. Instead, you should treat this data as being entirely transient, and take advantage of Laravel's database *seeding* capabilities to quickly and conveniently populate the database for you.

As an added bonus, you can use Laravel's database seeding capabilities for other purposes, including conveniently populating helper tables. For instance if you wanted users to provide their city and state when creating a new account, then you'd probably require the user to choose the state from a populated select box of valid values (Ohio, Pennsylvania, Indiana, etc.). These values might be stored in a table named `states`. It would be time consuming to manually insert each state name and other related information such as the ISO abbreviation (OH, PA, IN, etc). Instead, you can use Laravel's seeding feature to easily insert even large amounts of data into your database.

You'll seed a database by executing the following artisan command:

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

If you execute this command now, it will seem like nothing will happen. Actually, something *did* happen, it's just not obvious what. The `db:seed` command executes the file `database/seeds/DatabaseSeeder.php`, which looks like this:

```
<?php

use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        // $this->call(UsersTableSeeder::class);
    }
}
```

You'll use `DatabaseSeeder.php` to pre-populate, or *seed*, your database. The `run()` method is where all of the magic happens. Inside this method you'll reference other seeder classes, and when `db:seed` executes, the instructions found in those seeder classes will be executed as well. Inside this method you'll find an commented example call to a hypothetical class called `UsersTableSeeder`:

```
// $this->call('UsersTableSeeder::class');
```

The corresponding `UsersTableSeeder.php` file doesn't actually exist, but if it did it would be found in the `database/seeds/` directory. Let's create a seeder for adding a few events to the database. You can create the seeder skeleton using the `make:seed` command:

```
$ php artisan make:seed EventTableSeeder
Seeder created successfully.
```

You'll find the newly created `EventTableSeeder.php` file in `database/seeds/`. Update the file to look like this:

```
<?php

use App\Event;

use Illuminate\Database\Seeder;

class EventTableSeeder extends Seeder {

  public function run()
  {

    DB::table('events')->truncate();

    Event::create([
        'name' => "Laravel and Coffee",
        'city' => "Dublin",
        'venue' => "Cup of Joe",
        'description' => "Let's learn Laravel together!"
    ]);

    Event::create([
        'name' => "IoT and the Raspberry Pi",
        'city' => "Columbus",
        'venue' => "Columbus Library",
        'description' => "Weather monitoring with the Pi"
    ]);

  }

}
```

The `run` method begins by executing the Laravel Query Builder's `truncate` method (you'll learn more about the Query Builder later in this chapter). The `truncate` method will not only delete any records currently found in the `events` table, but it will also reset the table's auto incrementing primary key to its starting value. This is followed by two calls to the `create` method in which we pass along an array containing the model attributes and associated values.

Save this file and then replace the `DatabaseSeeder.php` line `$this->call('UserTableSeeder');` with the following lines:

```
$this->call('EventsTableSeeder');
```

Finally, run the seeder anew:

```
$ php artisan db:seed
Seeding: EventTableSeeder
```

Check your project database and you should see two new records in the `events` table!

Incidentally, you're not required to run your seed files via the `DatabaseSeeder` class. You can run a specific seeder by passing its class name along as an argument to the `db:seed` command like so:

```
$ php artisan db:seed --class=EventTableSeeder
```

Creating Large Amounts of Sample Data

Developers often make the classic mistake of not populating their development database with enough records to at least approximate what one might expect to see in the production environment. By simulating a realistic volume of data, you'll be able to account for issues and uncover problems not otherwise noticed when working with only a few records.

But repeatedly calling the `create` method hundreds or even thousands of times isn't going to be practical. Further, I certainly don't want to be bothered with dreaming up countless combinations of event names, venues, cities, and descriptions. Fortunately you can use the fantastic Faker library[55] (which is automatically included in Laravel projects) to easily create large amounts of sample data.

We'll use Faker to create fifty events. Modify the `EventTableSeeder`'s `run()` method to look like this:

```
class EventTableSeeder extends Seeder {

  public function run()
  {

    DB::table('events')->truncate();

    $faker = \Faker\Factory::create();

    foreach(range(1,50) as $index)
    {

        Event::create([
          'name'        => $faker->sentence(2),
          'city'        => $faker->city,
          'venue'       => $faker->company,
          'description' => $faker->paragraphs(1, true),
```

[55] https://github.com/fzaninotto/Faker

```
18          ]);
19
20      }
21
22    }
23
24  }
```

In this example a new instance of the `Faker` class is created. Next, a `foreach` statement is used to loop over `Event::create` fifty times, with each iteration resulting in Faker creating two random Lorem Ipsum[56]-style sentences consisting of two and four words, respectively.

Save the changes and run `php artisan db:seed` again. After the command completes, enter your database and you should see fifty records that look similar to the records found below:

```
1  mysql> select name, city, venue from events limit 4;
2  +-----------------------+-------------------+--------------------------+
3  | name                  | city              | venue                    |
4  +-----------------------+-------------------+--------------------------+
5  | Nisi modi amet.       | New Destinee      | Johnston, Herzog and Von |
6  | Et maiores.           | Otiliahaven       | Kovacek PLC              |
7  | Modi sit voluptatibus. | Cartershire      | Hauck-Hartmann           |
8  | Ut aliquam.           | South Hilbertside | Torp Ltd                 |
9  +-----------------------+-------------------+--------------------------+
```

The names and descriptions generated by Faker's `sentence` and `paragraphs` methods aren't going to look particularly realistic, but does it really matter? Our primary goal is to have enough data on hand to efficiently test features such as pagination, event favoriting, and event modification. For this purpose it doesn't particularly matter if the data "looks" real. Further, if you wanted to include a few realistic entries, use the `create` method as we did previous to add a few records prior to looping over the Faker-generated data. The possibilities here are limited only by your imagination!

You can also use Faker to generate random numbers, lorem ipsum paragraphs, male and female names, U.S. addresses, U.S. phone numbers, company names, e-mail addresses, URLs, credit card details, colors, and more! Be sure to check out the Faker documentation[57] for examples of these other generators.

Tinkering with the Event Model

As your models grow in complexity, you'll spend a lot of time experimenting with and testing functionality. One of the most convenient ways to do so is via the Tinker console (introduced in chapter 1). I find Tinker to be such an indispensable part of my development process that an active session is almost always open within a terminal tab. Begin by opening a new Tinker session:

[56] http://en.wikipedia.org/wiki/Lorem_ipsum
[57] https://github.com/fzaninotto/Faker

```
$ php artisan tinker
Psy Shell v0.8.14 (PHP 7.1.8 â€” cli) by Justin Hileman
>>>
```

Create a new `Event` object. To eliminate having to prefix your model names with the namespace, you can declare the namespace as demonstrated here:

```
>>> namespace App;
```

Next create a new instance of the `Event` class:

```
>>> $event = new Event;
=> App\Event {#789}
```

Next, assign an event name and description:

```
>>> $event->name = 'Laravel and Coffee';
=> "Laravel and Coffee"
>>> $event->city = 'Dublin';
=> "Dublin"
```

Now review the object's contents:

```
>>> $event
=> App\Event {#789
     name: "Laravel and Coffee",
     city: "Dublin",
   }
```

You can confirm the `$event` object's class name using PHP's `get_class()` function:

```
>>> get_class($event);
=> App\Event
```

Finally, have a look at the `Event` class' available methods using PHP's `get_class_methods()` function:

```
>>> get_class_methods($event)
=> [
     "__construct",
     "clearBootedModels",
     "fill",
     "forceFill",
     "newInstance",
     "newFromBuilder",
     ...
     "unguarded",
     "isFillable",
     "isGuarded",
     "totallyGuarded",
   ]
```

As you can see by these examples, Tinker provides a convenient means for frictionless experimentation with your project models. We'll return to Tinker throughout this book to introduce new concepts and fiddle with code.

Introducing Resourceful (RESTful) Controllers

Recall from the last chapter we created an `Events` controller, and inside it created two actions: `index` and `show`. The latter action even accepted an `$id` parameter, allowing IDs to be passed via the corresponding route in order to retrieve the event records stored in the database. In addition to demonstrating controller fundamentals, I wanted to give you a preview of a much more powerful and intuitive way to design your web applications.

We'll logically want to interact with the model within the HackerPair application in order to create, retrieve, manage, and even delete events. These fundamental tasks are so central to web applications that most popular web frameworks, Laravel included, implement *representational state transfer* (REST), an approach to designing networked applications that codify the way in which these tasks (create, retrieve, update, and delete) are implemented. Resourceful applications use the HTTP protocol and a series of well-defined URL endpoints to implement the seven actions defined in the following table:

HTTP Method	Path	Controller	Description
GET	/events	events#index	Display all events
GET	/events/create	events#create	Create a new event
POST	/events	events#store	Save a new event
GET	/events/:id	events#show	Display a event
GET	/events/:id/edit	events#edit	Edit a event
PUT	/events/:id	events#update	Update a event
DELETE	/events/:id	events#destroy	Delete a event

The `:id` included in several of the paths is a placeholder for a record's primary key (sound familiar?). For instance, if you wanted to view the event identified by the primary key `42`, then the URL path would look like `/events/42`. At first glance, it might not be so obvious how some of the other paths behave; for instance REST newcomers are often confused by the difference between `POST /events` and `PUT /events/:id`. Not to worry! We'll sort all of this out in the sections to come.

Creating a Resourceful Controller

Laravel natively supports REST-based routing (the Laravel developers refer to RESTful controllers as *resourceful controllers*, therefore herein I'll refer to them as such). You can use Laravel's `make:controller` command to create a resource controller:

```
$ php artisan make:controller EventsController --resource
Controller created successfully.
```

If you were following along in chapter 2 and already created an `Events` controller, you'll need to delete it before executing the above example while leaveing the corresponding view directory and routes intact (we'll address those in a moment).

Regardless of which generator you use, you'll find the generated controller in `app/Http/Controllers/EventsController.php`. Open the newly created file and you'll find the following code (comments removed for reasons of space):

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class EventsController extends Controller
{

    public function index()
    {
        //
    }

    public function create()
    {
        //
    }
```

```
19
20      public function store(Request $request)
21      {
22          //
23      }
24
25      public function show($id)
26      {
27          //
28      }
29
30      public function edit($id)
31      {
32          //
33      }
34
35      public function update(Request $request, $id)
36      {
37          //
38      }
39
40      public function destroy($id)
41      {
42          //
43      }
44
45  }
```

Laravel's resourceful controllers contain seven public methods, with each intended to correspond with an endpoint defined in the earlier table. But Laravel doesn't know you intend to use the newly generated controller in a RESTful fashion until the route definitions are defined. Open up the `routes/web.php` file and add the following line:

```
1  Route::resource('events', 'EventsController');
```

If you added the `Events` controller's `index` and `show` routes in the last chapter, replace those with the above statement.

Now if you try to access `http://hackerpair.dev/events` you'll be greeted with a blank page. This is because the `Events` controller's `index` action does not yet identify a view to be rendered! So let's create that view. If you didn't already do so in chapter 2, create a new directory named `events` inside `resources/views` (if this doesn't make any sense then please return to the previous chapter and read the relevant material). This is where all of the views associated with `Events` controller

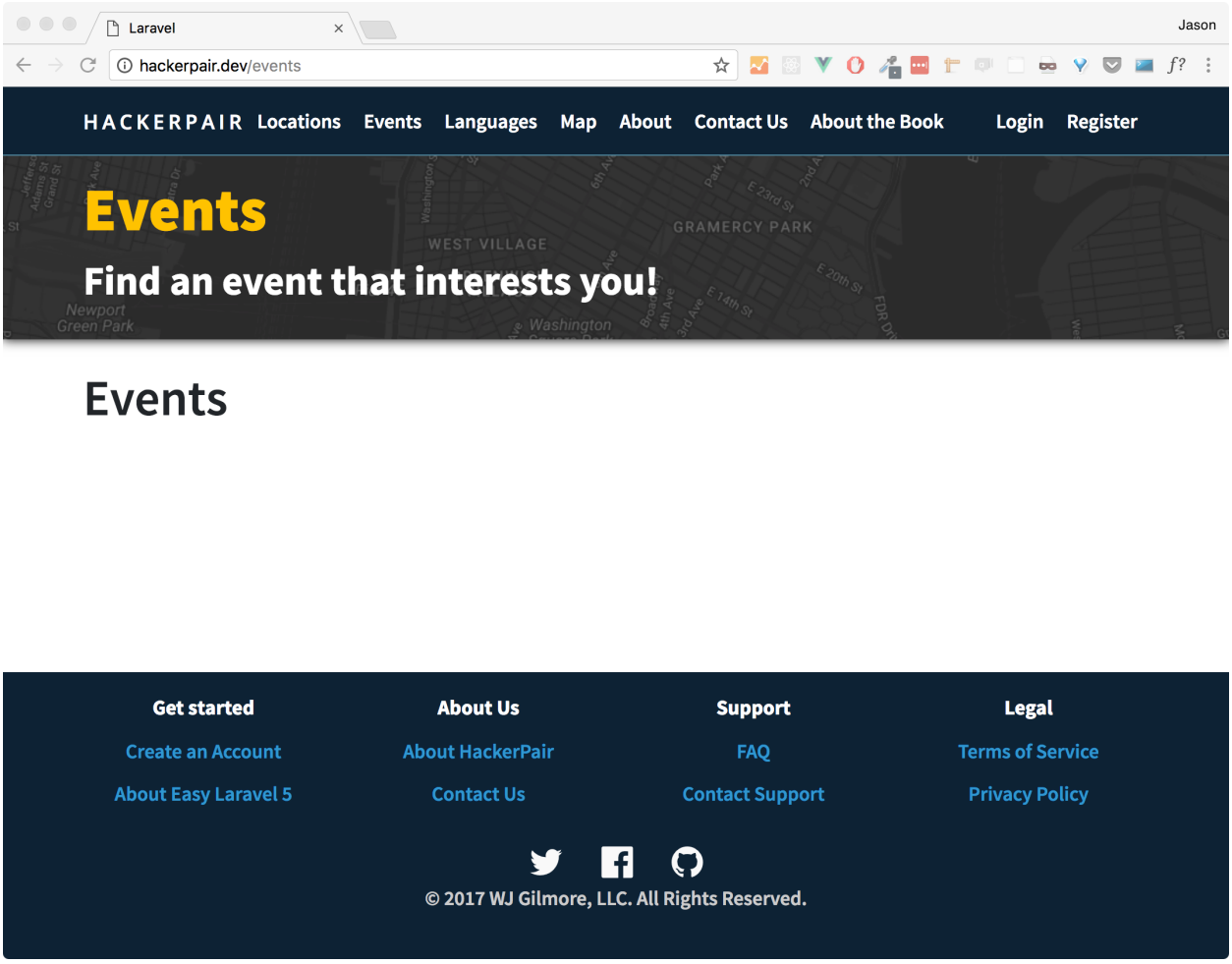
will be housed. Next, create a file named `index.blade.php`, placing it inside this directory. Add the following contents to it:

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

@section('content')

<h1>Events</h1>

@endsection
```

Next, open `app/Http/Controllers/EventsController` and modify the `index` action to look like this:

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

Save the file and navigate to `http://hackerpair.dev/events`. You should see the application layout (created in chapter 2) and the `h1` header found in `index.blade.php`. Here's what mine looks like, complete with the various UI improvements I made at the conclusion of the last chapter:



Serving a page from the resourceful Events controller

Congratulations, you've just implemented your first resourceful Laravel controller! Next, we'll integrate the `Event` model into the new controller.

Integrating a Model Into Your Controller

Now that you have a bit of experience interacting with a Laravel model, let's integrate an `Event` model into the `Events` controller. Return to the `index` action, which currently looks like this:

```
1  public function index()
2  {
3      return view('events.index');
4  }
```

If you recall from the earlier introduction a resourceful controller's `index` action is typically used to display a list of records. So let's use the `Event` model in conjunction with this action and

corresponding view to display a list of events. Begin by importing the `App\Event` into the controller. Strictly speaking you aren't obligated to do this but it will ultimately save some typing when invoking the class throughout the controller. The import should be placed at the very top of the controller alongside the other `use` statements:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use App\Event;

class EventsController extends Controller {

...

}
```

Next, modify the `index` action to look like this:

```
public function index()
{
    $events = Event::all();
    return view('events.index')->with('events', $events);
}
```

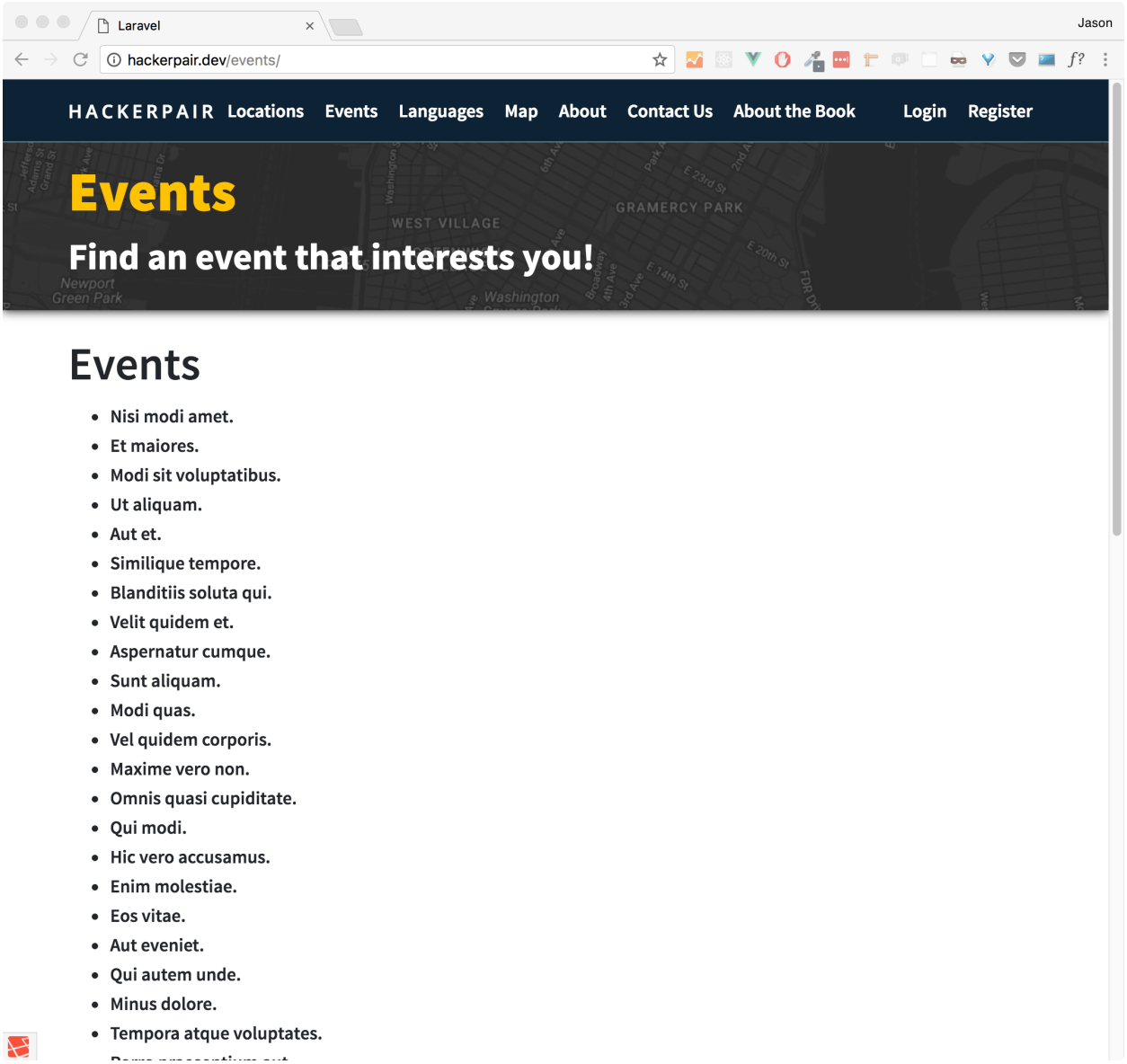
We're using the `all` method to retrieve all of the `Event` records found in the `events` table. This returns an object of type `Illuminate\Database\Eloquent\Collection` which is among other things iterable! We want to iterate over that collection of records in the view, and so `$events` is passed into the view.

Next, open the corresponding view (`resources/views/events/index.blade.php`), and modify the `content` section to look like this:

```
<h1>Events</h1>

<ul>
    @forelse ($events as $event)
        <li>{{ $event->name }}</li>
    @empty
        <li>No events found!</li>
    @endforelse
</ul>
```

The updated `index` view uses Blade's `forelse` method (introduced in chapter 2) to determine whether `$events` contains at least one element. If so, each event name in the collection will be displayed; otherwise an error message of sorts will be output. Each object's properties are exposed using PHP's standard object notation, meaning you can access an `Event` object's `name` property using `$event->name`.

Save the changes to your view, navigate to `/events` and you should see a bulleted list of all events found in the `events` table!



Displaying events

Paginating Events

The `all` method is certainly useful, but you probably noticed that with a meager 50 events in the database the results are already spilling off the page. While you'll want to provide users with a complete list of events, you'll typically want to *paginate* the results rather than dump all of them onto the same screen.

Database pagination is accomplished using a series of queries involving `limit` and `offset` clauses. For instance, to retrieve the first twenty events in batches of ten sorted by the `id` column you would execute the following queries (MySQL, PostgreSQL and SQLite):

```
select id, name from events order by id asc limit 10 offset 0
select id, name from events order by id asc limit 10 offset 10
```

Therefore when creating a pagination solution you would need to keep track of the current offset and limit values, the latter of which might be variable if you gave users the opportunity to adjust the number of items presented per page. Further, you would also need to create a user interface for allowing the user to navigate from one page to the next. Fortunately, pagination is a key feature of many web applications, meaning turnkey solutions are often incorporated into frameworks, Laravel included!

To paginate results, you'll use the `paginate` method:

```
$events = Events::paginate(10);
```

In this example we're overriding the default number of records retrieved (15), lowering the number retrieved to 10. Once retrieved, you can iterate over the collection using `@forelse` just as you would were pagination not being used. You'll want to make a slight modification to the view, adding the pagination ribbon:

```
{!! $events->links() !!}
```

This will create a stylized list of links to each available page, similar to the screenshot presented below.

Laravel's pagination ribbon

If you're using Bootstrap 4 you'll need to make a slight configuration change in order to achieve the same visual result as that presented in the above screenshot. Begin by executing the following command:

```
$ php artisan vendor:publish --tag=laravel-pagination
Copied Directory [/vendor/laravel/.../Pagination/resources/views] To
[/resources/views/vendor/pagination]
Publishing complete.
```

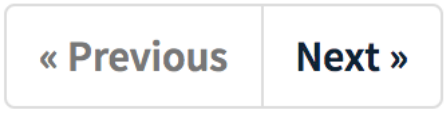
This command exported a variety of pagination control view partials to `resources/views/vendor/-pagination`. One of them is called `bootstrap-4.blade.php`. Pass that view partial name to the `links` method inside your `events/index.blade.php` view:

```
{!! $events->links('vendor.pagination.bootstrap-4') !!}
```

Using the Simple Paginator

If presenting a list of numbered links to each set of paginated results isn't important, and a simple set of left and right arrows will suffice, use the `simplePaginate()` method instead as you'll benefit from a slight performance improvement:

```
$events = Event::simplePaginate(10);
```



Laravel's simple pagination ribbon

Again, if you're using Bootstrap 4 then you'll need to run the aforementioned `vendor:publish` command as described, and then pass the `simple-bootstrap-4` view partial name into the `links` method like so:

```
{!! $events->links('vendor.pagination.simple-bootstrap-4') !!}
```

Implementing the Show Action

In the last chapter we implemented a `show` action inside the `Events` controller which demonstrated how to pass an ID from the URL into an action. You might have noticed the `show` action skeleton found in the generated resourceful controller looks identical to the version we manually created. But now that we have an `Event` model and underlying `events` table, we can use that ID to retrieve a record. Open the `Events` controller and update the show action to look like this:

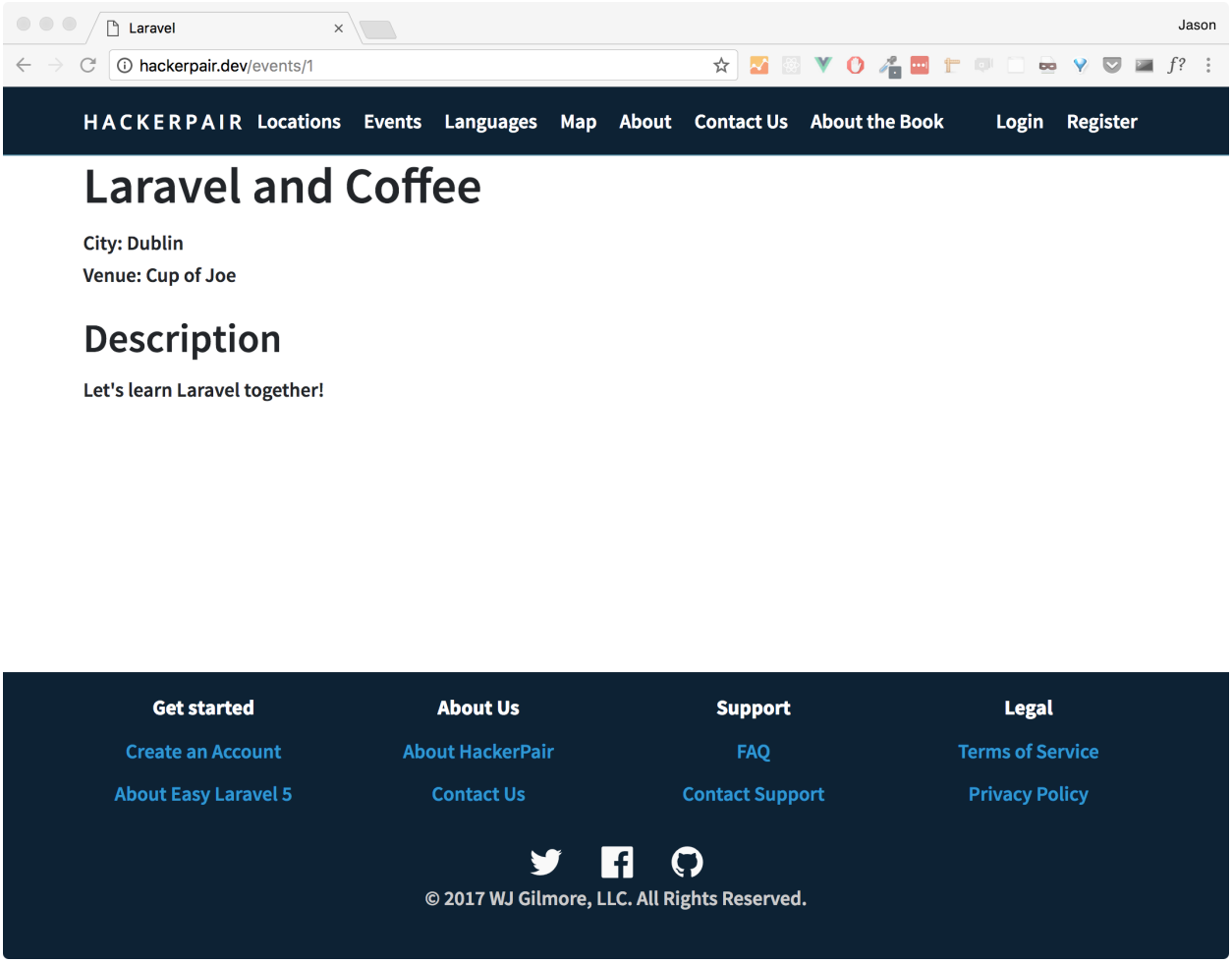
```
public function show($id)
{

    $event = Event::find($id);

    return view('events.show')->with('event', $event);

}
```

Eloquent's `find` method will retrieve a record associated with the provided ID. That record is then assigned to the `$event` variable which is subsequently passed into the view.

Next we'll create the corresponding view. Create a file named `show.blade.php`, placing it in the directory `resources/views/events`. Add the following contents to it:

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

@section('content')

<h1>{{ $event->name }}</h1>

<p>
City: {{ $event->city }} <br />
Venue: {{ $event->venue }}
</p>

<h2>Description</h2>
<p>
{{ $event->description }}
</p>

@endsection
```

After saving the changes, navigate to the `Events` controller's `show` URI (append an appropriate ID to `/events/` such as `/events/1`) and you should see output that looks something like this:



Displaying a specific event

Brilliant! We're now able to view more information about a specific event.

Saving Keystrokes with Route Model Binding

If you take a moment to peruse the resourceful `Events` controller you'll see that four of the seven actions accept an `$id` parameter. This is because the target record's ID is essential for retrieving a record (`show`), updating a record (`edit` and `update`), and deleting a record (`destroy`). In each of these actions we're expected to query the database in order to perform the desired task; that's a foregone conclusion. To save you the hassle of repetitively writing this code, Laravel offers a feature known as *route model binding*. When a controller is configured for route model binding, a model instance will be passed into the action rather than an ID.

To see route model binding in action, delete your existing `Events` controller, and create it anew using the following command:

```
$ php artisan make:controller EventsController --model=Event
```

Open the `Events` controller and you'll see the various `$id` parameters have been replaced with typehinted `$event` parameters. For instance here's what the `show` action looks like:

```
public function show(Event $event)
{
    //
}
```

Because the `$event` parameter will be automatically populated with the record associated with the ID passed along via the URL, we can just return the view and event like so:

```
public function show(Event $event)
{
    return view('events.show')->with('event', $event);
}
```

Because your project models will often be accompanied by a controller, you can generate model bound controllers at the same time as when the model is created:

```
$ php artisan make:model Event --controller --resource
Model created successfully.
Controller created successfully.
```

Obviously model bound controllers are preferred over having to repetitively retrieve the record, so its what we'll use when applicable for the remainder of the book.

Tweaking Your Eloquent Queries

Most Laravel queries are very straightforward in that they'll simply involve retrieving a record based on its primary key or some other filter, while others require more sophisticated approaches involving multiple parameters, grouping, and complex sorting. Fortunately Laravel offers an incredibly rich set of methods for querying data in a variety of fashions. In this section I'll demonstrate the many ways in which data can be retrieved from your application's database.

Selecting Specific Columns

For performance reasons you should to construct queries that retrieve the minimal data required to complete the desired task. For instance if you're constructing a view that only displays the event name and description, there is no reason to retrieve the `id`, `created_at`, and `updated_at` columns. You can specify the columns to be selected using the `select` method, as demonstrated here:

```
$ php artisan tinker
Psy Shell v0.8.14 (PHP 7.1.8 â€” cli) by Justin Hileman
>>> namespace App;
>>> $events = Event::select('name', 'city')->get();
=> Illuminate\Database\Eloquent\Collection {#855
 all: [
   App\Event {#856
     name: "Nisi modi amet.",
     city: "New Destinee",
   },
   ...
   App\Event {#905
     name: "Qui ratione quibusdam.",
     city: "Port Jefferey",
   },
 ],
}
```

Counting Records

To count the number of records associated with a given model, use the `count` method:

```
>>> Event::count();
50
```

You can also use the `count()` method within your view in conjunction with a collection to determine how many records have been selected:

```
$events = Event::all();
...
{{ $events->count() }} records selected.
```

Ordering Records

You can order records using the `orderBy` method. You'll use this method in conjunction with `get`. The following example will retrieve all `Event` records, ordered by name:

```
$events = Event::orderBy('name')->get();
```

You'll use the `get` method to retrieve records when using methods other than `all` or `find`, unless you're solely interested in the first item in that collection, in which case you can use `first`.

Laravel will by default sort results in ascending order. You can change this default behavior by passing the desired order (`ASC` or `DESC`) as a second argument to `orderBy`:

```
$events = Event::orderBy('name', 'DESC')->get();
```

You can order results using multiple columns by calling `orderBy` multiple times:

```
$events = Event::orderBy('created_at', 'DESC')
  ->orderBy('name', 'ASC')
  ->get();
```

This is equivalent to executing the following SQL statement:

```
SELECT * FROM events ORDER BY created_at DESC, name ASC;
```

Using Conditional Clauses

While the `find` method is useful for retrieving a specific record, you'll often want to find records using other attributes. You can do so using the `where` method. Suppose the `Event` model included a Boolean `published` attribute, intended to denote whether the event organizer has made the event public. You could use `where` to retrieve a set of completed events:

```
$publishedEvents = Event::where('published', '=', 1)->get();
```

Notice how the attribute, comparison operator, and value are passed into `where` as three separate arguments. This is done as a safeguard against attacks such as SQL injection. If the comparison operator is `=`, you can forego providing the equal operator altogether:

```
$publishedEvents = Event::where('published', 1)->get();
```

If you're using an operator such as `>` or `<`, you are logically required to expressly supply the operator. You can alternatively use the `whereRaw` method (without sacrificing security) to accomplish the same result:

```
$publishedEvents = Event::whereRaw('published = ?', 1)->get();
```

You can use `where` in conjunction with `!=` to retrieve records *not equal* to a given value:

```
$unpublishedEvents = Event::where('published', '!=', 1)->get();
```

Still other variations of the `where` method exist. For instance, you can use `whereBetween()` to retrieve records having a column value falling within a range:

```
$teenagers = User::whereBetween('age', [13, 19])->get();
```

Similarly, `whereNotBetween()` can be used to filter a range out of the results:

```
$adults = User::whereNotBetween('age', [13, 19])->get();
```

Incidentally, it's perfectly acceptable to chain `where*()` methods together to produce a compound conditional clause. The following example will retrieve a list of adult users who live in Ohio:

```
$adultsInOhio = User::whereNotBetween('age', [13, 19])
  ->where('state', 'Ohio')->get();
```

This will produce the following SQL:

```
select * from users where `age` not between 13 and 19 and `state` = 'Ohio'
```

Grouping Records

Grouping records according to a shared attribute provides opportunities to view data in interesting ways, particularly when grouping is performed in conjunction with an aggregate SQL function such as `count()` or `sum()`. Laravel offers a method called `groupBy` that facilitates this sort of query. Suppose you wanted to retrieve the years associated with all event creation dates along with the count of events associated with each year. You could construct the query in MySQL like so:

```
mysql> select year(created_at) as `year`, count(name) as `count`
    -> from events
    -> group by `year` order by `count` desc;
+--------------+
| year | count |
+------+-------+
| 2017 |   398 |
| 2016 |   247 |
| 2015 |   112 |
| 2014 |   92  |
| 2013 |   14  |
+---------------+
5 rows in set (0.00 sec)
```

This query can be reproduced in Laravel like so:

```
use DB;

...

$events = Event::select(DB::raw('year(created_at) as year'),
         DB::raw('count(name) as count'))
         ->groupBy('year')
         ->orderBy('count', 'desc')->get();
```

Another new concept was introduced with this example: `DB::raw`. Eloquent currently does not support aggregate functions but you can use Laravel's Query Builder interface in conjunction with Eloquent as a convenient workaround. The `DB::raw` method injects raw SQL into the query, thereby allowing you to use aggregate functions within `select`. Alternatively, you can use the `selectRaw` method to achieve the same outcome:

```
$events = Event::selectRaw('year(created_at) as year, count(name) as count')
->groupBy('year')
->orderBy('count', 'desc')
->get();
```

I'll talk more about Query Builder's capabilities in the later section, "Introducing Query Builder".

You can then iterate over the `year` and `count` attributes as you would any other:

```
<ul>
    @forelse ($events as $event)
        <li>{{ $event->count }} events created in {{ $event->year }}</li>
    @empty
        <li>No events found!</li>
    @endforelse
</ul>
```

You'll often want to group records in conjunction with a filter. For instance, what if you only wanted to retrieve a grouped count of events created after 2016? You could use `groupBy` in conjunction with `where`:

```
use DB;

$events = Event::select(
          DB::raw('year(created_at) as year'),
          DB::raw('count(name) as count'))
          ->groupBy('year')
          ->where('year', '>', '2016')->get();
```

This won't work because you're filtering on the non-aggregated field. What if you wanted to instead retrieve the same information, but only those years in which more than 500 events were created? It would seem you could use `group` in conjunction with `where` to filter the results, but as it turns out you can't use `where` to filter any column not calculated by an aggregate function, because `where` applies the defined condition *before* any results are calculated, meaning it doesn't know anything about the `year` alias at the time it attempts to perform the filter. Instead, when you desire to filter on the aggregated result, you'll use `having`. Let's revise the previous broken example to use `having` instead of `where`:

```
$events = Event::select(
          DB::raw('year(created_at) as year'),
          DB::raw('count(name) as count'))
          ->groupBy('year')
          ->having('year', '>', '2016')->get();
```

Limiting Returned Records

Sometimes you'll want to just retrieve a small subset of records, for instance the five most recently added events. You can do so using the oddly-named `take` method:

```
$events = Event::orderBy('id', 'desc')->take(5)->get();
```

If you wanted to retrieve a subset of records beginning at a certain offset you can combine `take` with `skip`. The following example will retrieve five records beginning with the sixth record:

```
$events = Event::orderBy('id', 'desc')->take(5)->skip(5)->get();
```

If you're familiar with SQL the above command is equivalent to executing the following statement:

```
SELECT * from events ORDER BY id DESC limit 5 offset 5;
```

Retrieving the First or Last Record

It's often useful to retrieve just the first or last record found in a collection. For instance you might want to highlight the most recently created event:

```
$event = Event::orderBy('id', 'desc')->first();
```

To retrieve a collection's last record, you can also use `first()` and reverse the order. For instance to retrieve the oldest event, you'll use the same snippet as above but instead order the results in ascending fashion:

```
$event = Event::orderBy('id', 'asc')->first();
```

Retrieving a Random Record

There are plenty of reasons you might wish to retrieve a random record from your project database. Perhaps a future version of HackerPair would highlight a random published event. Eloquent collections support the `inRandomOrder` method, which sorts the query results in random order. You can couple this method with `take(1)` and `first()` to retrieve a single random record:

```
$randomEvent = Event::where('published', true)->inRandomOrder()->take(1)->first()
```

This is equivalent to executing the following SQL:

```
select * from `events` where `published` = 1 order by RAND() limit 1
```

Determining Existence

If your sole goal is to determine whether a particular record exists, *without* needing to actually load the record if it does, use the `exists` method. For instance to determine whether a event associated with the city of Dublin exists, use the following statement:

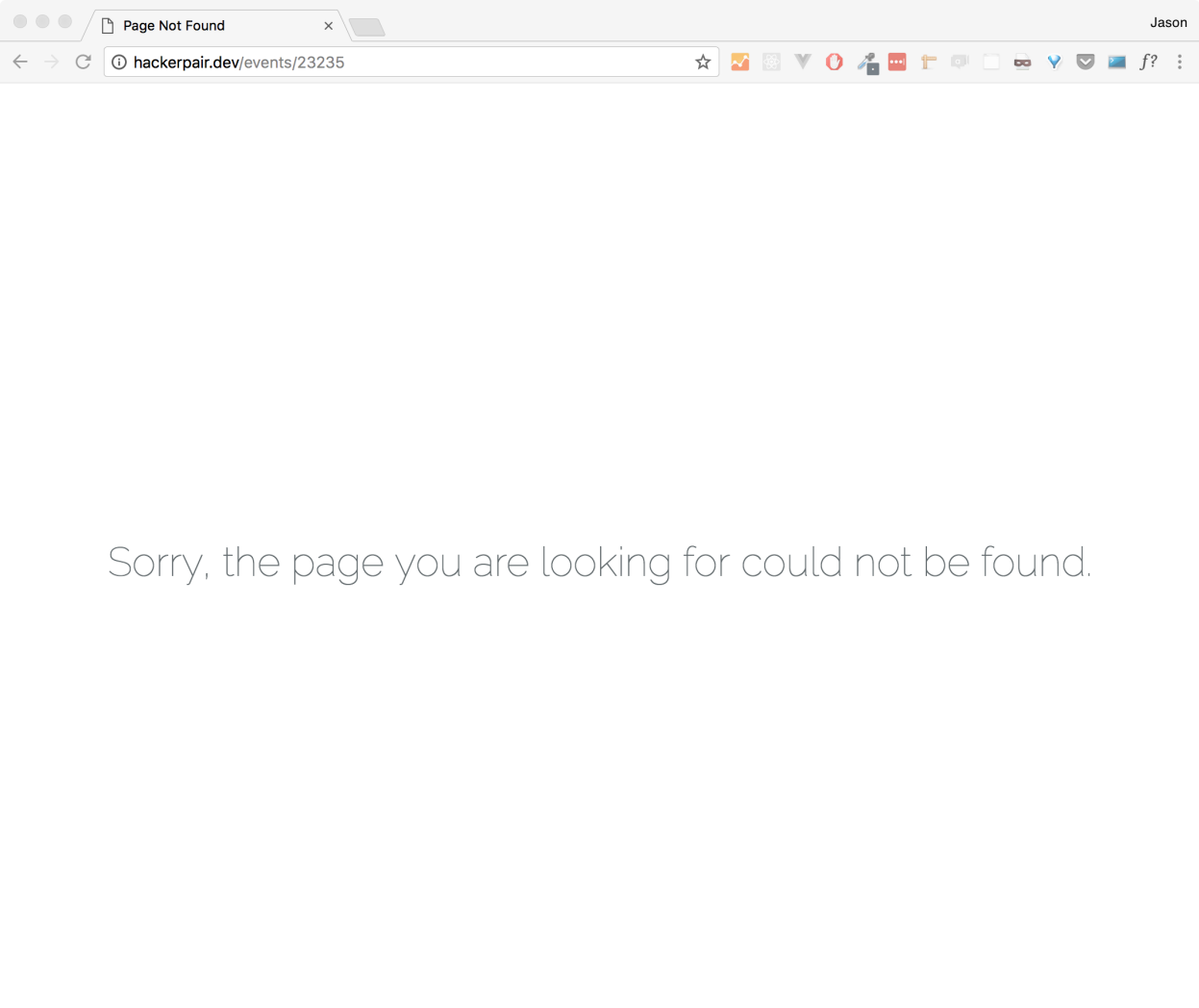
```
$eventExists = Event::where('city' , 'Dublin')->exists();
```

Using `exists` instead of attempting to locate a record and then examining the object or counting results is preferred for performance reasons, because `exists` produces a query that just counts records rather than retrieving them:

```
select count(*) as aggregate from `events`
where `city` = 'Dublin'
```

Gracefully Handling Requests for Nonexistent Records

So far our examples have been based on the premise that our queries are always successful. But what happens if the user attempts to access a record that doesn't exist, such as `/events/23245`? The `find` method would return a `null` value, meaning any attempt to subsequently display the event in a view would fail. If you're using route model binding, then Laravel will more gracefully throw an exception resulting in a 404 (page not found) error (see below screenshot).



Laravel's default 404 page

If you're not using route model binding, or are performing an additional query outside of the scope of route model binding, you can use the `findOrFail` method to cause the aforementioned error page to display should the record not be found:

```
1  $event = Event::findOrFail($id);
```

If the desired record is not found, an exception of type `ModelNotFoundException` will be thrown. If this exception is not caught following the failed query, the 404 error page will be displayed. But what if you wanted to handle the failure in another way? The easiest solution involves overriding the default 404 view. You can do this by creating a directory named `errors` inside `resources/views`, and inside the newly created directory create a view named `404.blade.php`. Populate the view with the desired error message, reload the page pointing to the nonexistent record, and you'll see the custom page.

Yet using the 404 page as a singular catch-all for such errors might not be desirable. You could instead create a customized response specifically for exceptions of type `ModelNotFoundException`. One way to handle this exception is by catching the exception directly within the action:

```
use Illuminate\Database\Eloquent\ModelNotFoundException;

...

public function show($id)
{

    try {

        $event = Event::findOrFail($id);
        return view('events.show')->with('event', $event);

    } catch(ModelNotFoundException $e) {

        return redirect()->route('events.index');

    }

}
```

This isn't very practical though, since you'd presumably have to catch the exception within every action containing a query. A much more scalable solution involves modifying the `app/Exceptions/Handler.php` class' `render` method. This method is responsible for converting exceptions into HTTP responses. It looks like this by default:

```
public function render($request, Exception $exception)
{
    return parent::render($request, $exception);
}
```

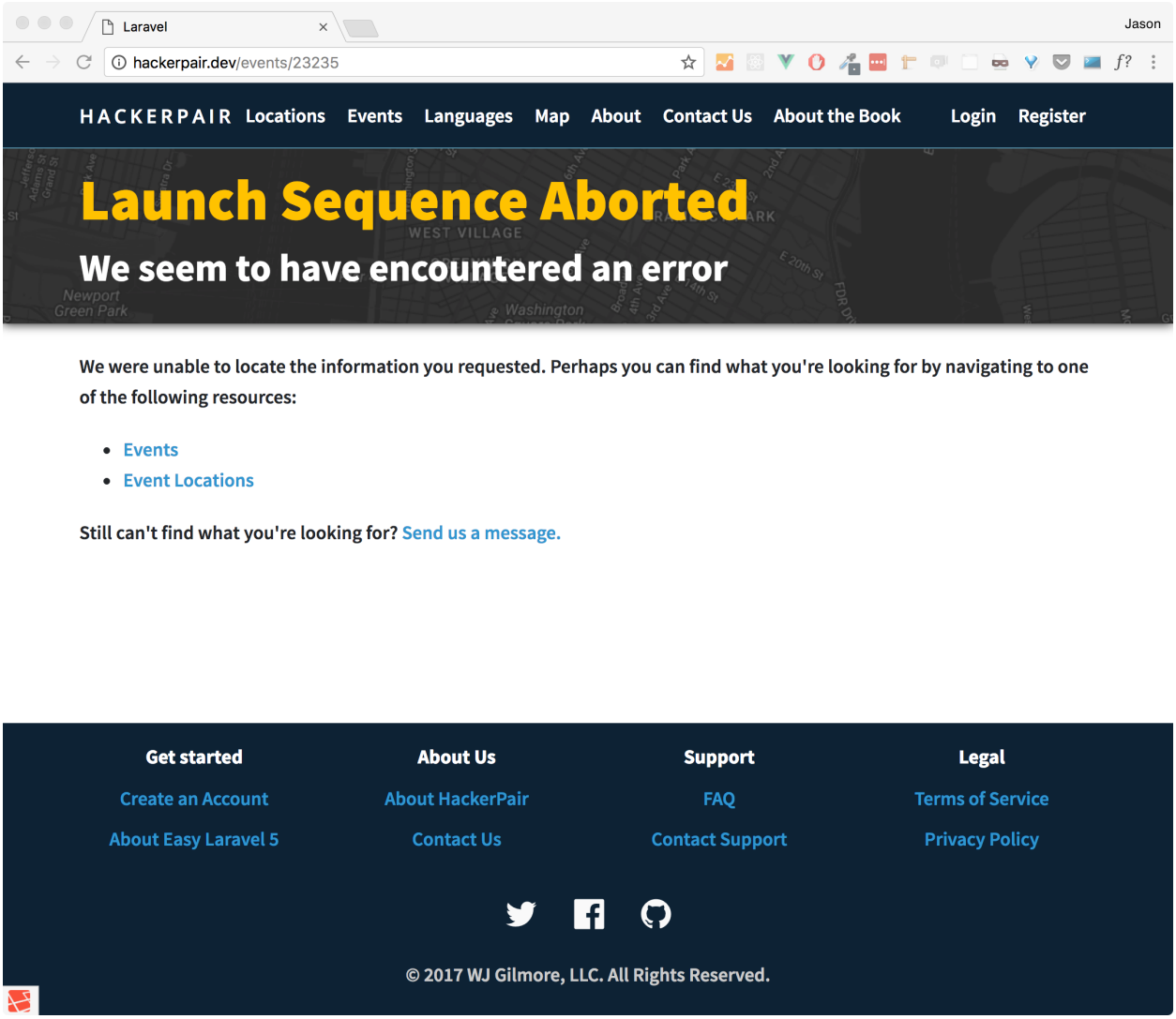
We can modify the method to determine whether an exception of type `ModelNotFoundException` has been thrown, and if so display a custom view:

```
public function render($request, Exception $exception)
{

    if ($exception instanceof ModelNotFoundException)
    {
        return response()->view('errors.model_not_found', [], 404);
    }

    return parent::render($request, $exception);
}
```

With this modification in place, you can remove the action-specific exception handler:

```
public function show($id)
{

    $event = Event::findOrFail($id);
    return view('events.show')->with('event', $event);

}
```

Finally, create a new directory inside `resources/views` named `errors`, and inside it create a view with a name matching that you used in the `render` method (I used `model_not_found.blade.php`). Here's what the HackerPair custom error page looks like now:



A custom error page

Introducing Query Builder

Chances are you'll be able to carry out most desired database operations using Eloquent. Even so, you'll occasionally want to exercise some of additional control over queries. Enter *Query Builder*, Laravel's alternative approach to querying your project database. Because the majority of your projects will be Eloquent-driven I don't want to dwell on Query Builder too much, but think this chapter would be incomplete without at least a brief introduction.

You can retrieve all of the records found in the `events` table using Query Builder like this:

```
use DB;

...

$events = DB::table('events')->get();
```

Prior to 5.3, Query Builder returned result sets in arrays. As of 5.3 these sets are returned as collections, meaning you can iterate over and interact with these results sets in precisely the same manner as described earlier in this chapter:

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

If you're looking for a specific record and want to search for it by ID, you can use `find`:

```
$event = DB::table('events')->find(42);
```

If you're only interested in retrieving the `name` column, there's no sense retrieving the notes and other columns. You can use `select` to limit the results accordingly:

```
$events = DB::table('events')->select('name')->get();
```

Finally, there are occasions where it makes more sense to directly execute raw SQL. You can do this using several different approaches. To select data, you can use `DB:select`:

```
$events = DB::select('SELECT * from events');
```

Unlike earlier examples, `DB::select` returns an array of objects (even in 5.3).

If for some reason you wanted to insert, update, or delete data using raw SQL, you can use the `DB::insert`, `DB::update`, and `DB::delete` methods, respectively:

```
DB::insert('insert into events (name, city) values (?, ?)',
  ['Laravel and Coffee', 'Dublin']);

DB::update('update events set published = 1 where id = ?', [42]);

DB::delete('delete from events where published = 0');
```

If you wanted to run SQL that isn't intended to interact with the data directly, perhaps something of an administrative nature, you can use `DB::statement`:

```
$events = DB::statement('drop table events');
```

As mentioned, this isn't intended to be anything more than a brief introduction to Query Builder. See the documentation[58] for a much more comprehensive summary of what's available.

Summary

This chapter covered an incredible amount of ground. You learned how to integrate a database into your Laravel application, manage schema migrations, seed the database with test data, and query the database using Eloquent and Query Builder. But this is really only the beginning in terms of what's possible when you combine Laravel and a powerful relational database. Let's move on to some more advanced model-related concepts that will really kick your application into high gear!

[58] http://laravel.com/docs/master/queries

Chapter 4. Customizing Your Models

In the last chapter we created an `Event` model for the sole purpose of interacting with the database through an object oriented interface. In practice, your application models will be used for a great deal more than a convenient bridge. In this chapter you'll learn how to customize models to enhance application security, create virtual attributes, extend their querying capabilities, and more. I'll also show you how to take advantage of a powerful feature known as *scopes* to reduce the amount of code you'd otherwise have to write to execute advanced queries throughout your application. The chapter concludes with a section on configuring Laravel's testing environment to test your models in a variety of ways.

Managing Model Dates and Times

Whenever you generate a migration intended to create a new table, Laravel will automatically include the following statement within the `Schema::create` body:

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

If you leave this statement in place, the newly created table will include `created_at` and `updated_at` fields, both of which are of type `timestamp`. Laravel will then automatically convert these fields to Carbon objects whenever they're retrieved from the database. Carbon is an incredibly useful PHP class created by Brian Nesbitt[59]. If you've seeded the `events` table, you can see this feature in action using Tinker:

```
1  $ tinker
2  Psy Shell v0.8.14 (PHP 7.1.8 â€” cli) by Justin Hileman
3  >>> namespace App;
4  >>> $event = Event::find(1)
5  >>> $event->created_at
6  => Illuminate\Support\Carbon {#819
7       +"date": "2017-11-29 04:01:38.000000",
8       +"timezone_type": 3,
9       +"timezone": "UTC",
10     }
```

As you can see from the Carbon documentation[60] this opens up all sorts of interesting possibilities. For instance you can retrieve just the event creation date (and not its time) using the `toDateString()` method:

[59] http://nesbot.com

[60] http://carbon.nesbot.com/docs/

```
>>> $event->created_at->toDateString()
=> "2017-11-29"
```

You can customize the date format using the `format` method:

```
>>> $event->created_at->format('F d, Y')
=> November 29, 2017
```

Logically you'll want to apply this capability to all of your temporal attributes. To do so, you'll identify these attributes in the model's `$dates` property. Open the `Event` model and add the following property to it:

```
protected $dates = [
  'created_at',
  'updated_at',
  'started_at'
];
```

In the HackerPair application the `events` table's `started_at` attribute is a `datetime` field which specifies the event's starting date and time.

Because we're overriding the default `$dates` property, we need to add `created_at` and `updated_at` to the array along with any other temporal fields we'd like to automatically convert into a Carbon object. With this in place, you're free to use Carbon's capabilities in conjunction with these other fields!

Defining Accessors and Mutators

Always remember Laravel models are just POPOs (Plain Old PHP Objects) that by way of extending the `Model` class have been endowed with additional capabilities. This means you're free to take advantage of PHP's object-oriented features to further enhance the model, including adding accessors (also known as getters), mutators (setters).

Defining Accessors

There's nothing stopping HackerPair users from creating event names which don't follow English language title capitalization standards. For instance a user might create an event named "let's learn a Little Laravel together". Ideally it would be presented as "Let's Learn a Little Laravel Together". We can create an `Event` model accessor which at least improves these titles to a version more closely approximating a grammatically correct version.

In Laravel accessors are encapsulated with methods using the naming convention `getPropertyAttribute`. Therefore to customize the `name` attribute's accessor we'll create an `Event` class instance method named `getNameAttribute`:

```
public function getNameAttribute($value)
{

    $ignore = ['a', 'and', 'at', 'but', 'for', 'in', 'the', 'to', 'with'];

    $name = explode(' ', $value);

    $modifiedName = [];

    foreach ($name as $word)
    {

        if (! in_array(strtolower($word), $ignore))
        {
            $modifiedName[] = ucfirst($word);
        } else {
            $modifiedName[] = strtolower($word);
        }

    }

    return join(' ', $modifiedName);

}
```

With this accessor defined, even if an event name were originally saved as "let's learn a Little Laravel together", with this accessor in place it will subsequently be retrieved as "Let's Learn a Little Laravel Together":

```
$ tinker
Psy Shell v0.8.14 (PHP 7.1.8 â€” cli) by Justin Hileman
>>> namespace App;
>>> $e = new Event;
=> App\Event {#789}
>>> $e->name = "let's learn a Little Laravel together"
=> "let's learn a Little Laravel together"
>>> $e->name
=> "Let's Learn a Little Laravel Together"
```

You can also define a *virtual accessors*, used to combine multiple attributes together. For instance, suppose a `User` model separates the user's name into `first_name` and `last_name` attributes. We'd like the option of easily retrieving the user's full name, which logically always consists of the first

name, followed by a space, followed by the last name. You can define an accessor to easily retrieve this virtual attribute:

```
class User extends Model {

  public function getFullnameAttribute()
  {
    return $this->first_name . " " . $this->last_name;
  }

}
```

Once saved, you can access the virtual `fullname` attribute as you would any other:

```
$user = User::find(12);
echo $user->fullname;
```

Defining Mutators

You'll use a *mutator* when you'd like to modify the value of an attribute prior to saving it to the database. User password storage is a particularly apt example of a mutator's utility. For security reasons the password only ever should be stored in the database using a hashed format (as you'll learn in chapter 7 Laravel conveniently takes care of all of this for you), meaning it's theoretically impossible to recreate the original value even when the hashed value is known. You want to be absolutely certain the password is only saved to the database using the chosen hashing algorithm, and therefore might create a mutator for the `password` attribute.

Laravel recognizes mutators when the method is defined using the `setAttributeNameAttribute` convention, meaning you'll want to create a method named `setPassword`:

```
class User extends Model {

  public function setPasswordAttribute($password)
  {
      $this->attributes['password'] = bcrypt($password);
  }

}
```

This example uses PHP's `bcrypt` function to generate a hash, accepting the plaintext password passed into the method, generating the hash, and assigning the hash to the `password` attribute. Here's an example:

```
$user = new User;
$user->password = 'blah';
echo $user->password;
$2y$10$e3ufaNvBFWM/SeFc4ZyAhe8u5UR/K0ZUc5IjCPUvOYv6IVuk7Be7q
```

To reiterate, this is just a hypothetical example. You don't actually have to create this mutator inside your `User` model because Laravel takes care of password hashing for you when the user creates an account (you'll learn more about this topic in chapter 7).

Creating and Using Instance Methods

Custom methods can greatly reduce the amount of logic otherwise cluttering your controller actions and views. For instance, the HackerPair `Event` model includes a `started_at` attribute which identifies the starting date and time in which the event is to take place. In the event detail view you might want to create a visual cue to bring special attention to any event occurring on the current date.

As you learned in the earlier section "Managing Model Dates and Times" it's possible to automatically convert a model's temporal attributes to a Carbon object. If you have a look at the HackerPair source code you'll see the `Event` model's `$date` property does in fact include the `started_at` column, meaning you can manipulate `started_at` using any of Carbon's great features. For instance you can use Carbon's `isToday` method inside the view to determine whether an event date is assigned the current date:

```
@if ($event->started_at->isToday())
    This event is occurring today!
@endif
```

Yet incorporating this logic in your actions and views can quickly become unwieldy. You can instead encapsulate it in a model's instance method like so:

```
class Event extends Model {

    public function occurringToday()
    {

        return $this->started_at->isToday();

    }

}
```

Then inside the view you can use the `occurringToday` method like so:

```
@if ($event->occurringToday())
    This event is occurring today!
@endif
```

The HackerPair models contain a number of convenient instance methods, so be sure to peruse the source code for ideas and inspiration!

Introducing Query Scopes

Instance methods are certainly useful but they are by definition applicable only to records that have already been retrieved from the database. What if you wanted to repeatedly use a query condition to limit the number of records returned? For instance, on http://hackerpair.com/events you'll only encounter events which have been expressly identified as "published" (visible to other users). In the last chapter you learned how to use the `where` method to query records according to some condition:

```
$publishedEvents = Event::where('published', '=', 1)->get();
```

But aren't we going to want to display only published events everywhere except within a user's event management interface. This means we'd need to use this `where` clause throughout the site, and worse, figure out how to override Laravel's default route model binding behavior. Fortunately there is a much more sane way to do this thanks to a Laravel feature known as *query scoping*.

Laravel supports two types of scopes, namely *global scopes* and *local scopes*. In this section I'll show you how to implement both.

Global Query Scopes

You'll want to use a global scope when the need arises to apply a query constraint by default, meaning calling for instance `Event::all()` would actually result in the following query being executed:

```
SELECT * FROM events WHERE `published` = 1;
```

You can implement global scopes in two ways: by managing the scope within a separate file, or by creating an anonymous global scope. I find the anonymous global scopes to be far more practical and so will leave it to you to review the Laravel documentation if you'd like to explore the former approach.

To implement an anonymous global scope within the `Event` model, add the following method to the class:

```
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Builder;

class Event extends Model
{

    ...

    protected static function boot()
    {

        parent::boot();

        static::addGlobalScope('published', function (Builder $builder) {
           $builder->where('published', '=', 1);
        });

    }

    ...

}
```

The `boot` method is where the magic happens. The `boot` method is defined in Laravel's `Model` class; it's used to initialize various model characteristics. We want to override this method to ensure a global scope is added to it, so we override the method while first making sure the parent method is called by executing `parent::boot()`. Subsequently we define the global scope using the `static::addGlobalScope` method:

```
static::addGlobalScope('published', function (Builder $builder) {
   $builder->where('published', '=', 1);
});
```

This method defines a new global scope named `published` which ensures `where published = 1` is appended to *every* select query initiated via the `Event` model. With the scope in place, return to the `/events` page and have a look the Laravel Debugbar's `Queries` tab. You'll see the following SQL statement:

```
select * from `events` where `published` = '1'
```

But wait a second. With this global scope in place, how are we going to provide users with a list of their unpublished events? Fortunately there is an easy solution. You can use the `withoutGlobalScopes` method to remove any global scopes otherwise attached to the model:

```
$events = Event::withoutGlobalScopes()->get();
```

Local Query Scopes

Global scopes are great when you want to effectively permanently attach a condition to a model's queries. But in many instances you want to selectively do so, adding a variety of conditions on a case-by-case basis. For instance instead of always applying the event publication condition, you could selectively apply it by creating a *local scope*. To create a local scope, you'll create an instance method in which the name begins with the term `scope`, like so:

```
public function scopePublished($query)
{
    return $query->where('published', 1);
}
```

With this in place, you can filter for published events like so:

```
$events = Event::published()->get();
```

You can also create local scopes which accept a parameter. These are known as *dynamic local scopes*. For instance, users will almost certainly want to search events according to zip code. The typical Eloquent query for such a purpose would look like this:

```
$events = Event::where('zip', $zipCode)->get();
```

Because we don't know what the desired zip code will be until the user provides it, you can create a scope which accepts the zip code as an input parameter:

```
public function scopeZip($query, $zip)
{
    return $query->where('zip', $zip);
}
```

The new scope can be used like so:

```
$zip = '43016';
$events = Event::zip($zip)->get();
```

Chaining Local Scopes

Scopes really begin to show their power when more than one is used at the same time. For instance, users might wish to filter events using a variety of conditions, such as zip code and maximum attendee count. The zip code scope has already been added to the `Event` model, so we just need to add the attendee count scope:

```
public function scopeAttendees($query, $maximum)
{
    return $query->where('max_attendees', $maximum);
}
```

With this scope in place, we can chain it and the zip code scope together like so:

```
$events = Event::zip(43016)->attendees(2)->get();
```

This will translate into the following query:

```
SELECT * FROM events WHERE zip = '43016' and max_attendees = '2';
```

Creating Sluggable URLs

Frameworks such as Laravel do a great job of creating user-friendly URLs by default, meaning the days of creating applications sporting ugly URLs like this are long gone:

```
http://hackerpair.com/events.php?id=42
```

Instead, Laravel will transform a URL like the above into something much more readable, such as:

```
http://hackerpair.com/events/42
```

While this may be an aesthetic improvement, the URL really isn't particular informative. After all, while you and I know both `42` is an integer value representing the primary key of a record found in the `events` table, the number conveys nothing of context to the user, much less to a search engine. but it would be much more practical to instead use a URL that looks like this:

```
http://hackerpair.com/events/laravel-hacking-and-coffee
```

This string-based parameter is known as a *slug*, and thanks to the eloquent-sluggable[61] package it's surprisingly easy to integrate sluggable URLs into your Laravel application. Begin by installing the package inside your application:

```
$ composer require cviebrock/eloquent-sluggable:^4.3
```

Because this particular version of eloquent-sluggable supports Laravel 5.5's package auto-discovery feature, there are no further installation steps and you're ready to begin creating sluggable URLs!

[61] https://github.com/cviebrock/eloquent-sluggable

Creating Sluggable Models

With the eloquent-sluggable package installed you'll need to update your models and underlying tables to enable the sluggable feature. This is fortunately incredibly easy to do. For instance, to add slugs to the `Event` model, modify it like so:

```
use Cviebrock\EloquentSluggable\Sluggable;

class Event extends Model {

    use Sluggable;

    public function sluggable()
    {
        return [
          'slug' => [
            'source' => 'name'
          ]
        ];
    }

    ...

}
```

There are several important changes to this model, including:

- The Sluggable class is imported at the top of the file
- Inside the class body you'll see the model uses the `Sluggable` trait.
- A method named `sluggable` is defined which identifies the database column where the slug should be saved, and the source column from where the slug should be created.

After saving the model changes you'll need to create a migration which adds the `slug` column to the table. If you haven't yet deployed your project to production you can just modify the migration used to create the sluggable table, adding a `string` column named `slug`. Alternatively you can create a separate migration like so:

```
$ php artisan make:migration add_slug_column_to_events_table \
> --table=events
Created Migration: 2017_11_22_225240_add_slug_column_to_events_table
```

Open the newly created migration and modify the `up` and `down` methods to look like this:

```
public function up()
{
    Schema::table('events', function (Blueprint $table) {
        $table->string('slug')->nullable();
    });
}

public function down()
{
    Schema::table('events', function (Blueprint $table) {
        $table->dropColumn('slug');
    });
}
```

After running the migration, the eloquent-sluggable package will *automatically* create the slugs for you any time a new record is added to the `events` table! For instance when I reran the events seeder (introduced in chapter 3) after adding the `slug` column to the `events` table, each event slug was automatically updated.

If you're adding this capability to an existing table that already contains data then you'll need to update each record. One of the easiest ways to do this is by entering the `Tinker` console, selecting all records and then saving them back to the database:

```
$ php artisan tinker
>>> namespace App;
>>> $events = Event::all();
>>> foreach ($events as $event) {
... $event->save();
... }
>>>
```

Creating Slug-based Queries

With the slugs in place, all you need to do is retrieve the desired record by slug rather than the integer ID. If you're using route model binding, you'll recall you no longer have to explicitly retrieve the desired record when an applicable action is called; Laravel will retrieve the ID in the URL and automatically retrieve the record for you provided you've injected the model instance into the action. However, Laravel will by default presume the parameter is associated with the integer-based `id` column, meaning you'll need to override this behavior so Laravel can instead query the model's underlying table using the `slug` column. This is done by adding the following method to your model:

```
public function getRouteKeyName()
{
    return 'slug';
}
```

Once the changes are saved, Laravel will instead use whatever column name is returned from this `getRouteKeyName` method.

If you're not using route model binding, another solution is available. The eloquent-sluggable package provides several useful helper method for performing slug-based queries. To use them you'll need to add the `SluggableScopeHelpers` trait to your model:

```
use Cviebrock\EloquentSluggable\Sluggable;
use Cviebrock\EloquentSluggable\SluggableScopeHelpers;

class Event extends Model
{

    use Sluggable, SluggableScopeHelpers;

    ...

}
```

With the trait in place, you can find a unique event record associated with the `laravel-hacking-and-coffee` slug using the `findBySlug` method:

```
$event = Event::findBySlug('laravel-hacking-and-coffee');
```

Alternatively, you can use the `findBySlugOrFail` method to the same effect as that described earlier in the chapter regarding `findOrFail`:

```
$event = Event::findBySlugOrFail('laravel-hacking-and-coffee');
```

As this section hopefully indicates, integrating sluggable URLs into your application is incredibly easy, and certainly improves the readability of your URLs!

Testing Your Models

Testing your models to ensure they are performing as desired is a crucial part of the application development process. Mind you, the goal isn't to test Eloquent's features; one can conclude Eloquent's capabilities are thoroughly tested by the Laravel developers. Instead, you want to focus on confirming proper behavior of features you incorporate into the application models, such as whether your model accessors and mutators are properly configured, and whether your instance methods and scopes are behaving as expected. With this in mind, let's take some time to investigate a few testing scenarios.

I'll presume you've successfully configured your Laravel testing environment as described in chapter 1. We do have a bit of additional configuration to do in order to integrate a database into this environment so let's take care of that first.

Configuring the Test Database

Because you'll want to test your application in conjunction with some realistic data you'll need to configure a test-specific database. This database will contain the very same database structure found in your development database (thanks to migrations), in addition to test data we'll create on a per-test basis. The easiest way to do so in Laravel is by adding test-specific environment variables to the `<php>` section of your `phpunit.xml` file:

```
<env name="DB_HOST" value="127.0.0.1" />
<env name="DB_PORT" value="3306" />
<env name="DB_DATABASE" value="test_hackerpair" />
<env name="DB_USERNAME" value="homestead" />
<env name="DB_PASSWORD" value="secret" />
```

Keep in mind you'll also need to create this database and the authentication credentials.

Automatically Rebuilding the Test Database

It is crucial for you to ensure that your database structure and test data are in a *known state* prior to the execution of each and every test, otherwise you're likely to introduce all sorts of uncertainty into the very situations you're trying to verify. One foolproof way to do this is by tearing down and rebuilding your test database structure prior to and following each test. To do so you can use the `RefreshDatabase` trait. If you open the `ExampleTest.php` test found in the `tests/Feature` directory you'll see the `RefreshDatabase` trait has been imported. All you have to do is use the trait as demonstrated below:

```
<?php

namespace Tests\Unit;

use Tests\TestCase;
use Illuminate\Foundation\Testing\RefreshDatabase;

class ExampleTest extends TestCase
{

    use RefreshDatabase

    public function testBasicTest()
    {
        $this->assertTrue(true);
    }
}
```

Creating a Model Factory

In addition to automated database migrations, Laravel supports *factories*. Factories are useful for generating sample data which can then be used in your tests. You'll find an example `User` model factory (the `User` model is introduced in chapter 7) in `database/factories/UserFactory.php`:

```
<?php

use Faker\Generator as Faker;

$factory->define(App\User::class, function (Faker $faker) {
    static $password;

    return [
        'name' => $faker->name,
        'email' => $faker->unique()->safeEmail,
        'password' => $password ?: $password = bcrypt('secret'),
        'remember_token' => str_random(10),
    ];
});
```

This factory uses the previously introduced Faker package to generate a placeholder name, e-mail address, password, and recovery token. Keep in mind you're free to set any column using a static

value, but tools such as Faker will save you quite a bit of hassle when you'd like to create a large number of sample records.

Let's create a new factory which will be used to test the `Event` model:

```
$ php artisan make:factory EventFactory
Factory created successfully.
```

A new file named `EventFactory.php` will be placed inside the `database/factories` directory. Open it and modify the contents to look something like this (this will probably vary depending upon exactly which columns you've added to the `events` table):

```
<?php

use Faker\Generator as Faker;

$factory->define(Model::class, function (Faker $faker) {
    return [
        'name'           => 'Laravel and Coffee',
        'published'      => 1,
        'start'          => '2017-12-01 15:00:00',
        'max_attendees'  => 3,
        'venue'          => 'City Coffee Shop',
        'city'           => 'Dublin',
        'description'    => "Let's drink coffee and learn Laravel together!"
    ];
});
```

With this factory available, next we'll create a new class for testing the `Event` model.

Creating Your First Model Test

For purely organizational purposes we'll manage the model-specific tests inside a directory named `Models` found in the `tests/Unit` directory. Create a new test class named `EventTest` using the following command:

```
$ php artisan make:test Models/EventTest --unit
```

You'll see a new directory named `Models` has been placed inside `tests/Unit`, and inside it a new file named `EventTest.php`. Update the file to look like this:

```
<?php

namespace Tests\Models;

use Tests\TestCase;
use Illuminate\Foundation\Testing\RefreshDatabase;

class EventTest extends TestCase
{

    use RefreshDatabase;

    public function testEventDateTimeFieldIsACarbonObject()
    {
        $event = factory(\App\Event::class)->make();
        $this->assertTrue(is_a($event->started_at, 'Illuminate\Support\Carbon'));
    }

}
```

At the beginning of this chapter you learned how to add temporal attributes to the model's `$dates` property to ensure they're automatically converted to Carbon objects when retrieved from the database. The `testEventDateTimeFieldIsACarbonObject` test ensures the `started_at` attribute wasn't mistakenly removed from `$dates`. You can run the test like so:

```
$ vendor/bin/phpunit tests/Unit/Models/EventTest.php
PHPUnit 6.4.3 by Sebastian Bergmann and contributors.

.                                   1 / 1 (100%)

Time: 558 ms, Memory: 16.00MB

OK (1 test, 1 assertion)
```

Great! So what else can we test? Earlier in the chapter we created a custom mutator which would correct an improperly capitalized event name. The mutator was called `getNameAttribute`, and whenever an event name was retrieved from the database the mutator would attempt to correct word capitalization, meaning an event named "have fun WITH the raspberry pi" would be converted to "Have Fun with the Raspberry Pi". Let's confirm this mutator is working as expected, and introduce another factory-feature along the way. Add the following test to the `EventTest` class:

```
public function testEventNameCapitalizationIsCorrect()
{

    $event = factory(\App\Event::class)->make(
        [
            'name' => "have fun WITH the Raspberry Pi"
        ]
    );

    $this->assertEquals($event->name, "Have Fun with the Raspberry Pi");

}
```

In addition to ensuring the event name casing is properly converted, this test introduces another useful factory-related feature: overriding factory attributes. In the above test we used the default `Event` factory, but overrode the `name` attribute to suit the specific needs of our test. Alternatively, if you plan on conducting a variety of capitalization-related tests, you can add a custom *state* definition to the `EventFactory`:

```
$factory->state(App\Event::class, 'incorrect_capitalization', [
    'name' => 'have fun WITH the raspberry pi',
]);
```

Then whenever you want to use an event with the malformed name, you can invoke the `Event` factory like so:

```
$event = factory(\App\Event::class)->states('incorrect_capitalization')->make();
```

The method is named `states` rather than `state` because it's possible to define and subsequently apply more than one state to a factory:

```
$event = factory(\App\Event::class)->states('city', 'zip')->make();
```

Persisting Factories

Previous examples used the `make` method to generate new instances of a factory. For some tests though, you'll want to persist the factory to the database and subsequently retrieve it later in the test. To do so you can use the `create` method:

```
$event = factory(\App\Event::class)->create();

...

$event = Event::find(1);
```

We'll repeatedly return to the `make` and `create` methods, as well as factories, throughout the remainder of this book.

Summary

Your march towards Laravel proficiency has once again reached another milestone now that you know how to customize models to meet your project's specific needs. Yet so far we've been focused exclusively on retrieving data. What about inserting, modifying, and deleting it? Not to worry as the entire next chapter is devoted exclusively to this topic!

Chapter 5. Creating, Updating, and Deleting Data

The last two chapters offered quite a few examples, but all were focused on the task of retrieving data from the database. Yet other than inserting data using seeders, we haven't touched at all upon the inserting data into the database, let alone modifying and deleting it. This topic's time has come! In this chapter you'll learn all about various approaches to data creation, update, and deletion, and how to customize your models to reflect a variety of desired behaviors pertaining to these tasks.

In the second part of this chapter we'll return to the `Events` controller first introduced in chapter 3, implementing the remaining actions (`create`, `store`, `edit`, `update`, and `destroy`). You'll also learn how to use flash notifications to inform users regarding the outcome of their requests.

Inserting Records Into the Database

Laravel offers a few different approaches to creating new records. The first involves the `save` method. You'll first create a new instance of the desired model, update its attributes, and then execute the `save` method:

```
1  $event = new Event;
2  $event->name = 'Coffee and Laravel';
3  $event->venue = 'The Mocha Factory';
4  $event->save();
```

Easy as that. Presuming your underlying table incorporates the default `id`, `created_at` and `updated_at` fields, Laravel will automatically update the values of these fields for you.

You can alternatively use the `create` method, simultaneously setting and saving the model attributes. But there's a catch here, and it's important you see this in action so let's start a new Tinker session and try using `create`:

```
$ tinker
>>> namespace App;
>>> $event = Event::create(['name'  => 'Coffee and Laravel',
... 'venue' => 'The Mocha Factory'] );
Illuminate\Database\Eloquent\MassAssignmentException with message 'name'
>>>
```

The call to `create` failed and a `MassAssignmentException` was thrown, because allowing a record to be created by mass-assigning values via an array could present a security risk. For instance, an improperly configured web form handler could allow an attacker to inject additional fields into a form and send them along with the expected fields. We'll talk more about such security hazards in chapter 7, so for the moment just understand that if you did want to allow mass-assignment of some fields by way of the `create()` method, you'll need to identify the fields which are allowed to be assigned in this fashion via the associated model's `$fillable` property, as demonstrated here:

```
class Event extends Model
{

    protected $fillable = [
        'name',
        'venue'
    ];

}
```

If the model's underlying table consisted of a large number of fields and you knew which ones were *not* allowed to be mass-assigned, you could instead use the `$guarded` property:

```
class User extends Model
{

    protected $guarded = ['is_admin'];

}
```

This means that all fields in the `users` table are mass-assignable *except* for `is_admin`, which might be used as a simple solution for determining whether a user can gain access to an application's administrative features.

Other Approaches to Record Creation

The `create` and `save` method examples presented above are based on the presumption those records don't already exist in the database. But what if you wanted to prevent records having certain duplicate attributes from being persisted? You can instead use the `firstOrCreate` method, which will first query the database for records assigned the provided attribute values. If there is no match, the record will be created, otherwise the existing record will be retrieved:

```
$event = Event::firstOrCreate(
    [
        'name' => 'Coffee and Laravel'
    ]
);
```

At this point you either have a newly created record, or an existing record assigned he name `Coffee and Laravel`. In any case, you're free to change for instance the venue and then save the changes back to the database:

```
$event->venue = 'Starclucks';
$event->save().
```

You could save a step if the intended goal is to either retrieve the existing record or create a new one with the name and additional attributes:

```
$event = Event::firstOrCreate(
    ['name' => 'Coffee and Laravel'],
    [
        'venue' => 'Starclucks',
        'city'  => 'Dublin'
    ]
);
```

If you instead just want to create a new model instance (and not immediately persist it) if a record matching the provided attribute isn't found, use `firstOrNew()`:

```
$event = Event::firstOrNew(['name' => 'Coffee and Laravel']);
```

Updating Existing Records

Users will logically want to update events, for instance perhaps tweaking the event name or description. To do so, you'll typically retrieve the desired record using its primary key, update the attributes as necessary, and use the `save` method to save the changes:

```
$event = Event::find(14);
$event->venue = 'Starclucks';
$event->save();
```

If your intent is to update a record if it exists or create a new record if no match is found, you might consider using `updateOrCreate`. Like `firstOrCreate`, it allows you to specify an attribute argument separately from the values you'd like to create or update depending upon whether a record is found:

```
$event = Event::updateOrCreate(
    ['name' => 'Coffee and Laravel'],
    [
        'venue' => 'Starclucks',
        'city'  => 'Dublin'
    ]
);
```

The first array defines the attributes used to determine whether a matching record exists, and the second array identifies the attributes and values which will be inserted or updated based on the outcome. If the former, then the attributes found in the first array will be inserted into the new record along with the attributes found in the second array.

Deleting Records

To delete a record you'll use the `delete` method:

```
$event = Event::find(12);
$event->delete();
```

You can optionally consolidate the `find` and `delete` commands using the `destroy` method:

```
Event::destroy(12);
```

Soft Deleting Records

In many cases you won't ever actually want to truly remove records from your database, but instead annotate them in such a way that they are no longer displayed within the application. This is known as *soft deletion*. Laravel natively supports soft deletion, requiring just a few configuration changes to ensure a model's records aren't actually deleted when `delete` or `destroy` are executed. As an example let's modify the `Event` model to support soft deletion. Begin by creating a new migration that adds a column named `deleted_at` to the `events` table:

```
$ php artisan make:migration add_soft_delete_to_events \
> --table=events
Created Migration: 2017_12_01_184402_add_soft_delete_to_events
```

Next open up the newly created migration (found in the `database/migrations` directory), and modify the `up` and `down` methods to look like the following:

```
public function up()
{
    Schema::table('events', function(Blueprint $table)
    {
        $table->softDeletes();
    });
}

public function down()
{
    Schema::table('events', function(Blueprint $table)
    {
        $table->dropColumn('deleted_at');
    });
}
```

Save the changes and run the migration:

```
$ php artisan migrate
Migrating: 2017_12_01_184402_add_soft_delete_to_events
Migrated:  2017_12_01_184402_add_soft_delete_to_events
```

After the migration has completed you'll next want to open up the target model and use the `SoftDeletes` trait:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;

class Event extends Model {

```

```
10      use SoftDeletes;
11
12      protected $dates = [
13          'created_at',
14          'deleted_at',
15          'started_at'
16          'updated_at'
17      ];
18
19  }
```

While not strictly necessary, adding the `deleted_at` attribute to the `$dates` array as demonstrated above will cause any returned `deleted_at` values to be of type `Carbon`. As introduced in chapter 4, adding timestamps to the `$dates` property is useful because you can use convenient Carbon methods such as `toFormattedDateString()` and `toDateTimeString()` to easily present formatted dates and times, not to mention manipulate date/time values in fantastically useful ways. See the Carbon docs[62] for more information.

After saving these changes, the next time you delete a record associated with this model, the `deleted_at` column will be set to the current timestamp. Any record having a set `deleted_at` timestamp will not be included in any retrieved results, thereby seemingly having been deleted. There are plenty of practical reasons why you might want to at some point include soft deleted records in your results (for instance giving users the ability to recover a previously deleted record). You can do so using the `withTrashed` method:

```
1  $events = Event::withTrashed()->get();
```

Implementing the Event Controller's Remaining Resourceful Actions

In chapter 3 we created the project's `Events` controller, and implemented the `index` and `show` actions. These are only two of the seven actions included by default in every new resourceful controller. The remaining five are responsible for inserting new data (`create` and `store`), updating existing data (`edit` and `update`), and deleting data (`destroy`). In this section you'll learn how to implement these actions, and along the way will learn how to create form interfaces in Laravel.

Creating New Events

Resourceful controllers provide the `create` and `store` actions which work together to insert a new record into the database. The `create` action is responsible for providing the interface used to populate

[62] http://carbon.nesbot.com/docs/

the payload which will be submitted to `store`, which is responsible for inserting the payload into the database.

Returning to the `Events` controller first created in chapter 3, you'll find skeletons for these two actions already in place:

```
public function create()
{
    //
}

public function store(Request $request)
{
    //
}
```

When defining the purpose of the `create` action in this section's opening paragraph, I was purposefully cryptic about exactly how the payload is generated, because it could be generated using any manner of mechanisms. In reality, this mechanism will typically be an HTML form. Therefore the `create` method will have the sole role of serving a view containing this form. Modify `create` to look like this:

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

The `resources/views/events` directory should already from chapter 3, so the only other step you'll need to take is creating an empty file inside this directory named `create.blade.php`. What goes inside this file is an entirely different matter compared to the views we've created thus far, insomuch that it warrants its own section.

Creating the Event Creation Form

A form-based view isn't any different from other views, except for the obvious inclusion of the HTML form. While it is certainly possible to hand-code the form's HTML markup, I instead strongly suggest taking advantage of a Laravel package named LaravelCollective/HTML, which was first introduced in chapter 2. This package provides a variety of useful view helpers which really go a long way towards tidying up your form markup.

If you installed LaravelCollective/HTML back in chapter 2, the only additional step you need to take to use the form-specific features is to open the `config/app.php` file and add the following line to the bottom of the `$aliases` array:

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

If you haven't yet installed the package and want to follow along, please return to chapter 2 for the installation instructions.

Once installed, you'll be able to integrate a well-formatted form with remarkably little work. For instance, to create a new text field, you'll use the `Form::text` helper:

```
{!! Form::text('name', null,
    [
        'class'       => 'form-control input-lg',
        'placeholder' => 'PHP Hacking and Pizza'
    ])
!!}
```

When rendered to the browser, the text field will look like this:

```
<input
    placeholder="PHP Hacking and Pizza"
    name="name"
    type="text"
    value=""
    id="name"
    class="form-control input-lg"
>
```

Notice how the helper is enclosed within `{!! !!}` delimiters rather than the usual `{{ }}` delimiters. This is because HTML will be output to the browser. Since the `{{ }}` delimiters will escape HTML code for security reasons, you need to manually override this safeguard using Laravel's supported `{!! !!}` delimiters.

The first `Form::text` argument is the string assigned to the `input` element's `name` attribute. This value will also be assigned to the `id` attribute unless you expressly assign a value to `id` in the array passed along as the third argument. The second argument, currently set to `null`, can be used to set the form field's `value` attribute. When set to `null`, a blank value will be used.

You're not required to format the `Form::text` arguments as I've done here. This is purely for organizational purposes.

You can place a label above the text field using the `Form::label` helper. For instance the following example combines a label and text field inside a Bootstrap `form-group` DIV:

```
<div class="form-group">
    {!! Form::label('name', 'Event Name', ['class' => 'control-label']) !!}
    {!! Form::text('name', null,
        [
            'class'       => 'form-control input-lg',
            'placeholder' => 'PHP Hacking and Pizza'
        ])
    !!}
</div>
```

The various fields comprising a form will be enclosed within `<form>...</form>` tags. The opening `<form>` tag will define the form's `action` and `method` attributes. The LaravelCollective/HTML package includes helpers for these opening and closing tags:

```
{!! Form::open(['route' => 'events.store'], ['class' => 'form']) !!}

...

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

If the `method` attribute is not specified, `Form::open` will default to `POST`, which is what we want to use when submitting *non-idempotent* data through a web form. Further, the `route` attribute is just one of several ways to identify the form action; you could for instance use `url` and instead identify a URI. I prefer to stick to using route aliases whenever possible and so have demonstrated that ability here.

If you need a refresher on web form concepts such as the difference between `GET` and `POST`, please see Appendix A. Web Form Fundamentals.

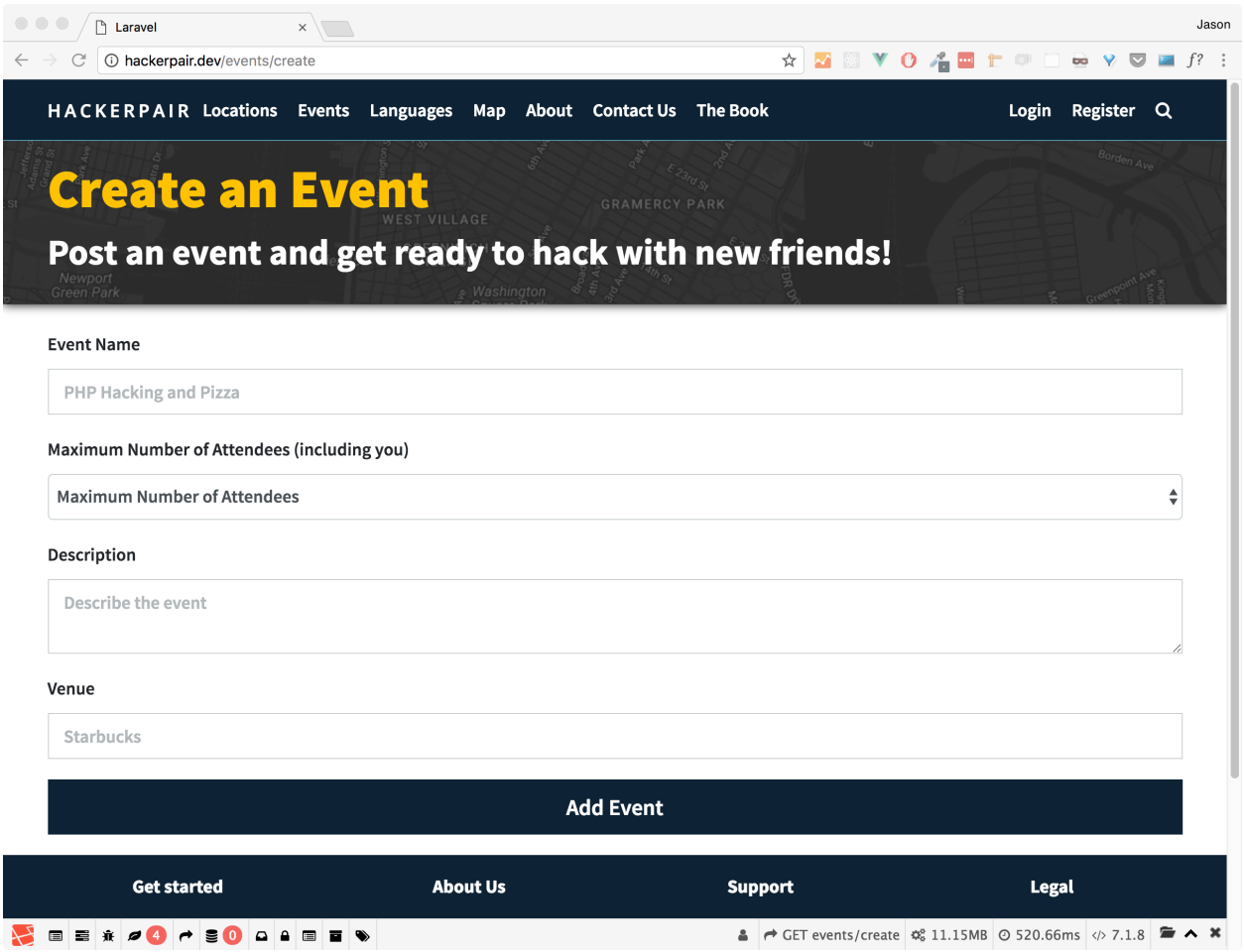
Here's a simplified example of the event creation form found in the HackerPair project's `resources/views/events/create.blade.php` view:

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

@section('content')

    <div class="row">

        <div class="col">

            {!! Form::open(['route' => 'events.store'], ['class' => 'form']) !!}
```

```
10
11              <div class="form-group">
12                  {!! Form::label('name', 'Event Name',
13                      ['class' => 'control-label'])
14                  !!}
15                  {!! Form::text('name', null,
16                      [
17                          'class' => 'form-control input-lg',
18                          'placeholder' => 'PHP Hacking and Pizza'
19                      ])
20                  !!}
21              </div>
22
23              <div class="form-group">
24                  {!! Form::label('max_attendees', 'Maximum Number of Attendees',
25                      ['class' => 'control-label'])
26                  !!}
27                  {!! Form::select('max_attendees', [2,3,4,5], null,
28                      [
29                          'placeholder' => 'Maximum Number of Attendees',
30                          'class' => 'form-control input-lg'
31                      ])
32                  !!}
33              </div>
34
35              <div class="form-group">
36                  {!! Form::label('description', "Description",
37                      ['class' => 'control-label'])
38                  !!}
39                  {!! Form::textarea('description', null,
40                      [
41                          'class' => 'form-control input-lg',
42                          'placeholder' => 'Describe the event'
43                      ])
44                  !!}
45              </div>
46
47              <div class="form-group">
48                  {!! Form::submit('Add Event',
49                      [
50                          'class' => 'btn btn-info btn-lg',
51                          'style' => 'width: 100%'
```

```
52                      ])
53                  !!}
54              </div>
55
56              {!! Form::close() !!}
57
58          </div>
59
60      </div>
61
62  @endsection
```

With the above form rendered to the browser, it will more or less look like that found in the following screenshot.



Creating a new Event

With the form in place and the `create` action updated, it's time to update the `store` action to process

any form data it receives.

Updating the Event Controller's Store Action

With the event creation form in place and pointing to the `events.store` route, all that remains is to update the `Event` controller's `store` action to process the form contents. In this case, processing the form contents means saving it the data to the database, as demonstrated in the below revised `store` method:

```
use App\Event;

...

public function store(Request $request)
{

    $event = Event::create([
        $request->input()
    ]);

    flash('Event created!')->success();

    return redirect()->route('events.show')->with('event', $event);

}
```

The `$request` object passed into the `store` action will contain quite a bit of information about the incoming request, including the form data. If you're using the model's `$fillable` property to allow for mass assignment, then creating a new record is as easy as using the model's `create` method and referencing the `$request` object's `input` method.

If you're not using `$fillable`, you can alternatively create a new model instance, set attributes, and execute the `save` method:

```
$event = new Event;
$event->name = $request->name;
$event->max_attendees = $request->max_attendees;
$event->description = $request->description;
```

You'll also often see this form data retrieved using the `$request` object's `all`, `get` or `input` methods, or through direct object dereferencing. For instance the following alternatives will retrieve the submitted event name:

```
$event->name = $request->name;
$event->name = $request->get('name');
$event->name = $request->input('name');
```

Once the event has been created, users are redirected to the newly created event's profile page. Prior to doing so I've created what's known as a *flash notification*. Flash notifications can provide users with useful feedback regarding the outcome of a request. In the next section we'll take a short detour so I can show you how to integrate this great feature into your application.

The examples in this chapter demonstrate the most simplistic approach to accepting and processing form data. In the real world, you *must* validate user input before blindly inserting it into the database. Read that last sentence twice. This is such an important matter that I devote the entire next chapter to data validation. Please for the love of programming carefully read the next chapter before deploying a forms-driven Laravel application to production.

Integrating Flash Notifications

Flash notifications contain information which will exist only for the duration of the subsequent HTTP request. They're particularly useful for providing users with feedback regarding the outcome of a request. Laravel supports flash notifications natively but a fair bit of additional work is required to stylize these notifications so I rely on a third-party package named laracasts/flash[63] to do the hard work for me. Once installed, you can create and display flash messages with just two lines of code.

To install laracasts/flash, navigate to your project's root directory and run the following command:

```
$ composer require laracasts/flash
```

Next, open `config/app.php` and add the following line to the `providers` array:

```
Laracasts\Flash\FlashServiceProvider::class,
```

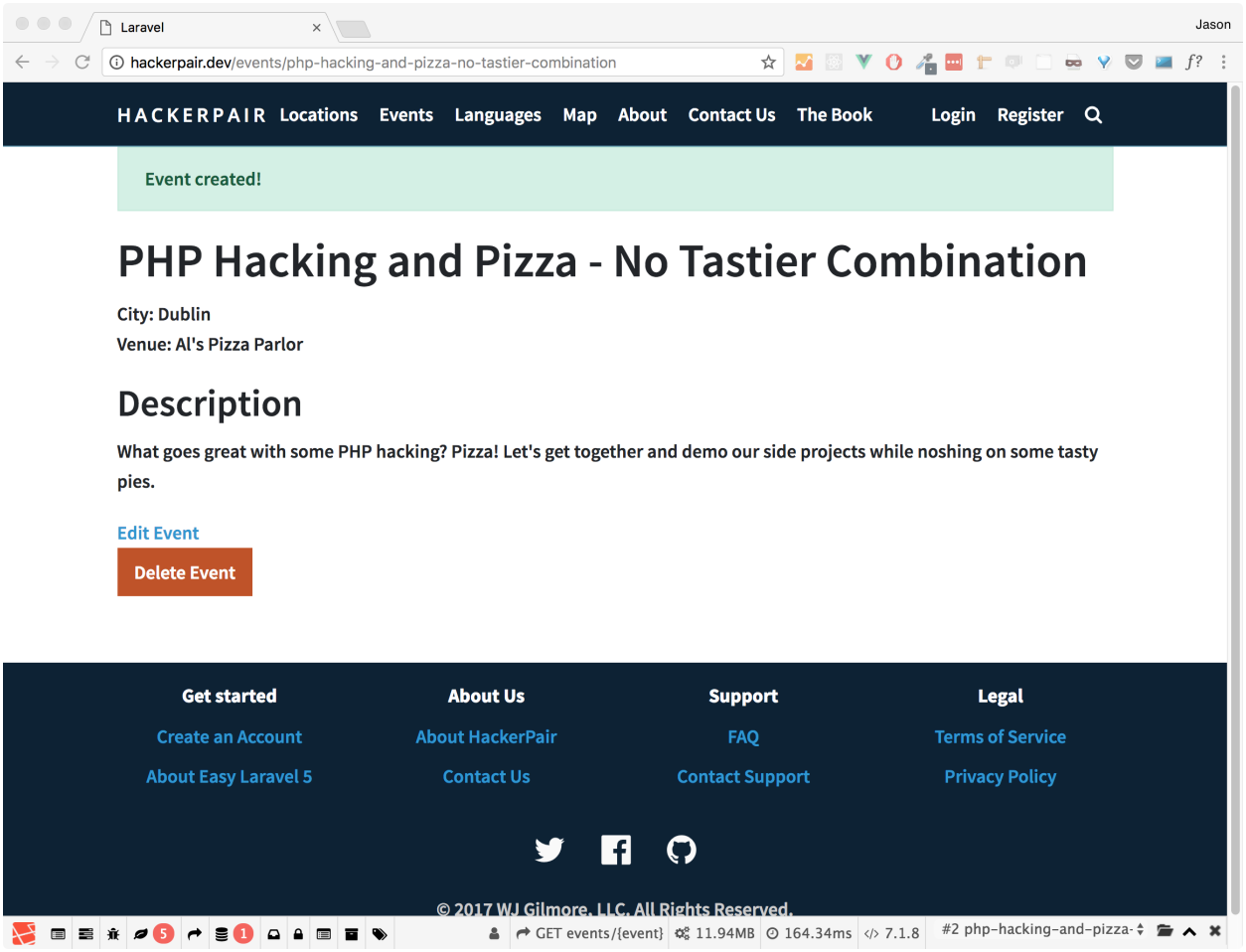
After saving the changes you only need to add a single new line to your layout file so the package knows where to render the flash messages:

[63] https://github.com/laracasts/flash

```
<div class="container">
    @include('flash::message')
    @yield('content')
</div>
```

With the flash message location defined, you're free to use the `flash` helper inside your controllers to define flash messages just as I did in the previous example.

The following screenshot presents an example of what the `success` flash message used in the earlier example looks like when rendered to the browser.



Displaying a flash message

Plenty of other message types are available, including `error`, `warning`, and `important`. You can additionally configure the package to instead present the messages as an overlay. Be sure to check out the package's GitHub page for more details.

Updating an Event

Users will understandably occasionally wish to change event details, so you'll need to provide a mechanism for updating an existing event. In many ways this feature's implementation is similar to that used for the event creation feature, with a few important differences. For starters, just as resourceful creation requires two actions (`create` and `store`), updates require two actions (`edit` and `update`). Open the `Events` controller and you'll see these two action method skeletons are already in place:

```
public function edit(Event $event)
{
    //
}

public function update(Request $request, Event $event)
{
    //
}
```

Just for clarification, if you are not using route model binding then these actions will accept different input parameters:

```
public function edit($id)
{
    //
}

public function update(Request $request, $id)
{
    //
}
```

The `edit` action is responsible for serving the form (which is filled in with the existing list's data), and the `store` action is responsible for saving the updated form contents to the database. Notice how both actions accept as input an `$id`. This is the primary key of the list targeted for modification. If you recall from chapter 3's REST-related discussion, these two actions are accessed via (in the case of the `Event`) controller `GET /events/:id/edit` and `PUT /lists/:id`, respectively. If the `PUT` method is new to you, not to worry because Laravel handles all of the details associated with processing `PUT` requests, meaning all you have to do is construct the form and point it to the `update` route. Let's take care of this next.

Creating the Event Edit Form

The form used to update a record is in most cases practically identical to that used to create a new record, with one very important difference. Instead of `Form::open` you'll use `Form::model`:

```
{!! Form::model($event,
  [
   'route' => ['events.update', $event->id],
   'class' => 'form'
  ]) !!}

...

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

The `Form::model` method binds the enclosed form fields to the contents of a model record. Additionally, be sure to take note of how the list ID is passed into the `events.update` route. This record is passed into the view like you would any other:

```
public function edit(Event $event)
{

   return view('events.edit')->with('event', $event);

}
```

When you pass a `Event` record into the `Form::model` method, it will bind the values of any attributes to form fields with a matching name. Let's create the form, beginning by creating a new view named `edit.blade.php` and placing it in the `resources/views/events` directory. Then add the following contents to the view:

```
{!! Form::model($event,
  [
    'method' => 'put',
    'route' => ['events.update', $event->id],
    'class' => 'form'
  ]
) !!}

<div class="form-group">
    {!! Form::label('name', 'Event Name') !!}
```

```
    {!! Form::text('name', null,
      ['class' => 'form-control']) !!}
</div>

<div class="form-group">
    {!! Form::label('description', 'Event Description') !!}
    {!! Form::textarea('description', null,
      ['class' => 'form-control']) !!}
</div>

<div class="form-group">
    {!! Form::submit('Update Event', ['class' => 'btn btn-primary']) !!}
</div>
{!! Form::close() !!}
```

Take special note of the form's method declaration. The `put` method is declared because we're creating a REST-conformant update request. After saving the changes to the `Events` controller and `edit.blade.php` view, navigate to the list edit route, being sure to supply a valid list ID (e.g. `/events/2/edit`) and you should see a populated form! Remember if you are using the sluggable URLs as introduced in the last chapter you'll instead pass along the slug, such as `events/ex-eum-consequatur/edit`.

You'll typically link events to the edit form, meaning you won't have to worry about whether the ID or slug is included in the URL. Presuming you're using the Laravel Collective package, then generating the appropriate link is as easy as using the `link_to_route` helper:

```
{{ link_to_route('events.edit', 'Edit Event', ['event' => $event])}}
```

In cases where the form used to create and edit a record are identical in every fashion except for the use of `Form::open` and `Form::model`, consider storing the form fields in a partial view and then inserting that partial into the `create` and `edit` views between the form opener and `Form::close` method.

Updating the Event Controller's update Action

With the `edit` action and corresponding view in place all that remains is to update the `update` action:

```
public function update(Request $request, Event $event)
{

    $event->update(
        $request->input()
    );

    return redirect()
        ->route('events.edit', $event)
        ->with('message', 'Event updated!');

}
```

Despite this action containing remarkably few lines of code, there are a number of nuances worth addressing. First, I'm taking the easy way out in terms of performing the update by passing all of the form fields into the `update` method using the `$request` object's `input()` method. Keep in mind that doing so will only result in those fields identified in the `Event` model's `$fillable` property being updated. Any field passed along from the form which is not identified within `$fillable` will simply be ignored. Therefore if you've decided to not include any fields in `$fillable` then you'll need to modify the `update` method's input to look like this:

```
$event->update([
  'name'        => $request->name,
  'description' => $request->description
]);
```

Deleting Events

When using resourceful controllers the `destroy` action is responsible for deleting the record. This action is by default only accessible via the `DELETE` method, as indicated when running `route:list`:

```
$ php artisan route:list

...
+--------+----------------+----------------+-----------------------------+
| Method | URI            | Name           | Action                      |
+--------+----------------+----------------+-----------------------------+
| DELETE | events/{event} | events.destroy | ...EventsController@destroy |
+--------+----------------+----------------+-----------------------------+
```

The `DELETE` method requirement means you can't just create a hyperlink pointing users to the `evenys.destroy`, because hyperlinks use the `GET` method. Instead you'll typically use a form with a stylized button to create the appropriate link, as demonstrated below:

```
{!! Form::open(
  [
    'route' => ['events.destroy', $event],
    'method' => 'delete'
  ]) !!}
    {!! Form::submit('Delete Event', ['class' => 'btn btn-danger']) !!}
{!! Form::close() !!}
```

Notice how the `Form::open` method's `method` attribute is overridden (the default is `POST`) to instead use `DELETE`. The form's `route` attribute identifies the `events.destroy` route as the submission destination, passing in the event object. When submitted, the `Events` controller's `destroy` action will execute, which looks like this:

```
public function destroy(Event $event)
{

    $event->delete();

    return redirect()
        ->route('events.index')
        ->with('message', 'The event has been deleted!');

}
```

Conclusion

In many ways, this chapter forms the crux of much of your time spent with Laravel, because so many applications involve creating, updating, and destroying data. Be sure to master these features and you'll save a tremendous amount of time and aggravation in future projects.

Chapter 6. Validating User Input

In the last chapter you learned how to create interfaces for inserting and updating events. There was however a significant omission: the submitted data was never validated! Although proper use of Laravel's Eloquent and Query Builder interfaces will go a long way towards avoiding many potential catastrophes associated with attacks such as SQL injection, it is still your responsibility to ensure user input falls within the desired set of constraints before inserting it into the database or performing further processing.

Fortunately, Laravel's team has put a great deal of thought into security, offering a number of features useful for validating data. In this chapter you'll learn all about the two most prominent such solutions, namely validators and form requests. I'll also demonstrate a number of different solutions for displaying error messages for user consumption.

Introducing Validation Rules

User-submitted data is often expected to conform to a set of pragmatic constraints. For instance, if a user is asked for his age, we don't expect him to enter `643` Nor do we expect a valid zip code to be `AG&$3`. Yet users often can't be counted on to submit acceptable input, and so it is incumbent upon you to prevent inconsistent and nonsensical data from ever reaching the database or other processing destinations. One easy way to do so involves incorporating Laravel's *validation rules* into your workflow.

Laravel supports dozens of validation rules capable of examining everything from string length to an attachment's MIME type. What's more, they are so easily integrated into your controllers that there's really no excuse for not doing so. Returning to the event insertion example from the last chapter, recall the example form prompted the user to specify an event name, number of attendees (ranging from two to five), and a description. Suppose for formatting reasons you wanted to limit the event name length to at least `10` characters but less than `50` characters, and do not want the user to be able to increase the event attendance count to more than five attendees (or worse, do something silly such as set the count to `-42`). In its current form the `Event` controller's `store` action sets no such constraints:

```
public function store(Request $request)
{

    $event = Event::create([
        $request->input()
    ]);

    flash('Event created!')->success();

    return redirect()->route('events.show')->with('event', $event);

}
```

The user is free to make the event name as long as he desires (at least until the `name` column's `VARCHAR` maximum of 65,535 characters is met presuming you're using a MySQL database), and a user with even rudimentary technological sophistication can use a browser inspector to change the available select box values to anything he pleases. Fortunately, we can use validators to limit such tomfoolery:

```
public function store(Request $request)
{

    $request->validate([
        'name'          => 'required|string|min:10|max:50',
        'max_attendees' => 'required|integer|digits_between:2,5',
        'description'   => 'required|string'
    ]);

    $event = Event::create(
        $request->input()
    );

    flash('Event created!')->success();
    return redirect()->route('events.show', ['event' => $event]);

}
```

Thanks to the `$request` object's convenient `validate` method, we can pass along an associative array containing the name of each form field and its associated validators. Each validator is separated by a vertical pipe. For instance the `name` field will be validated using four validators:

- `required`: The value is required.
- `string`: The value must be a string.

- `min:10`: The value must consist of at least 10 characters.
- `max:50`: The value must consist of no more than 50 characters.

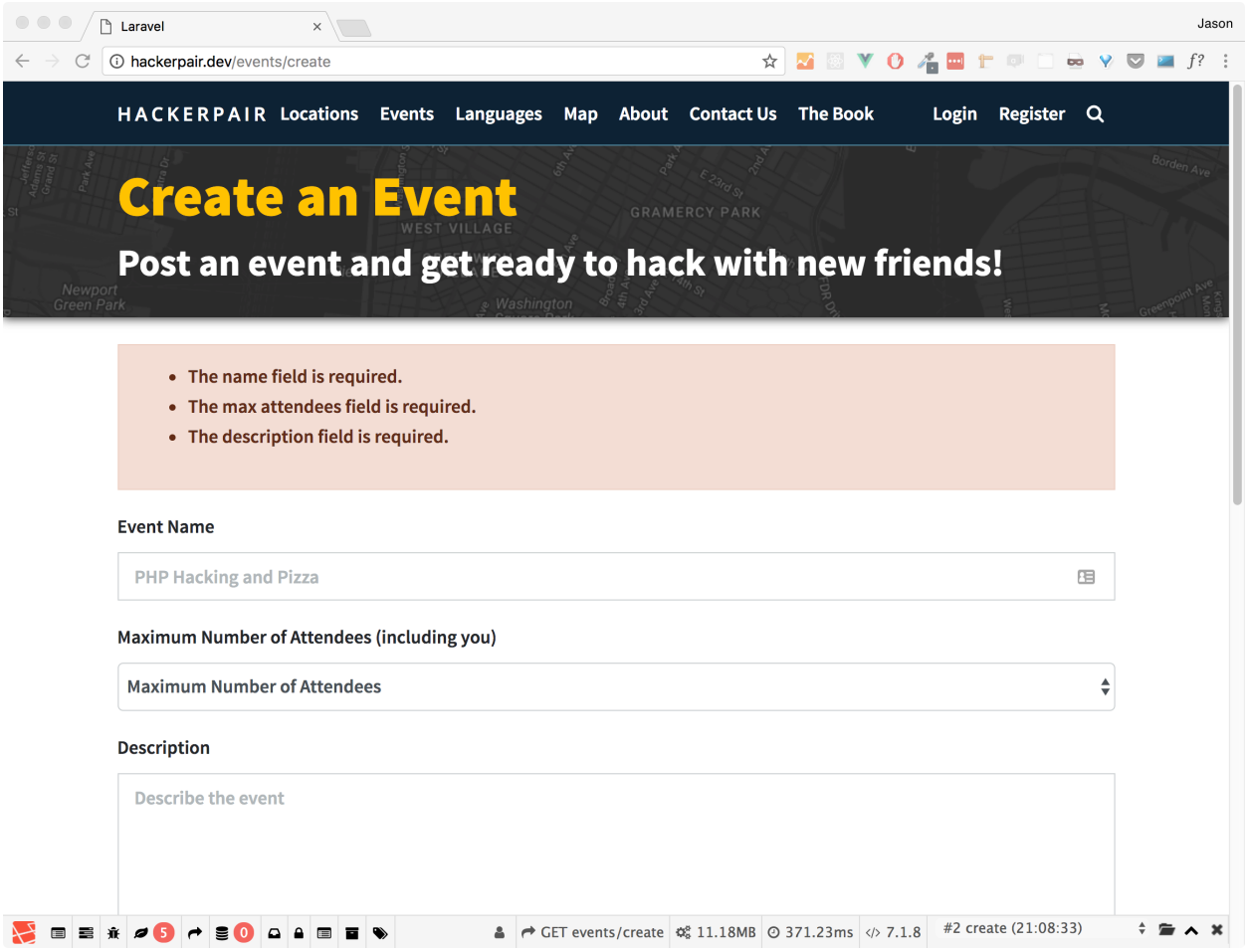
The `max_attendees` field is similarly locked down in that it is required, must be an integer, and must be a value between 2 and 5.

The `description` field is the least locked down of the bunch in that it is only required an must be a string.

Another great convenience afforded by Laravel validators is what happens should any of the validators fail; if any fail, the user will automatically be redirected back to the form. The only additional step you'll need to take is telling Laravel where to display any provided error messages and how you'd like them to be formatted. These errors are populated within an array named `$errors` which is automatically made available to the view. For instance, the relevant part of HackerPair's `app.blade.php` layout view looks like this:

```
<div class="container">

    @include('flash::message')

    @if ($errors->any())
        <div class="alert alert-danger">
            <ul>
                @foreach ($errors->all() as $error)
                    <li>{{ $error }}</li>
                @endforeach
            </ul>
        </div>
    @endif

    @yield('content')

</div>
```

After making these changes, try loading and submitting the event creation form without actually touching any of the fields. You should see a series of error messages above the form such as those presented in the following screenshot.



Validating user input

The validators introduced in this section hardly scratch the surface in terms of what's available. Be sure to consult the Laravel documentation[64] for a complete summary.

Customizing Error Messages

Laravel provides standard error messages for each validator, meaning you don't have to be bothered with dreaming up your own. However, it is possible to override the default messages with very little additional work. You'll just create an array which contains the desired error message for each validator, and then pass that array into a custom validator instance:

[64] https://laravel.com/docs/master/validation

```
use Illuminate\Support\Facades\Validator;

...

public function store(Request $request)
{

    $rules = [
        'name'          => 'required|min:10|max:50',
        'max_attendees' => 'required|integer|digits_between:2,5',
        'description'   => 'required'
    ];

    $messages = [
        'required' => 'Please provide an event :attribute',
        'max_attendees.required' => 'What is the maximum number of
          attendees allowed to attend your event?',
        'name.min' => 'Event names must consist of at least 10 characters',
        'name.max' => 'Event names cannot be longer than 50 characters',
        'max_attendees.digits_between' => 'We try to keep events cozy,
          consisting of between 2 and 5 attendees, including you.'
    ];

    Validator::make($request->input(), $rules, $messages)->validate();

    $event = Event::create(
        $request->input()
    );

    flash('Event created!')->success();
    return redirect()->route('events.show', ['event' => $event]);

}
```

The validation rules are defined just as we did previously. The only difference is I'm encapsulating them in a separate array for code organization's sake. Next up are the custom messages, found in the `$messages` array. What's really great about this particular feature is the ability to fine tune each validation message to suit the field. For instance, the first array element defines a blanket custom `required` error message for all fields. But because the maximum attendees field is named `max_attendees`, were the user to not specify the maximum number of attendees the error message would look like this:

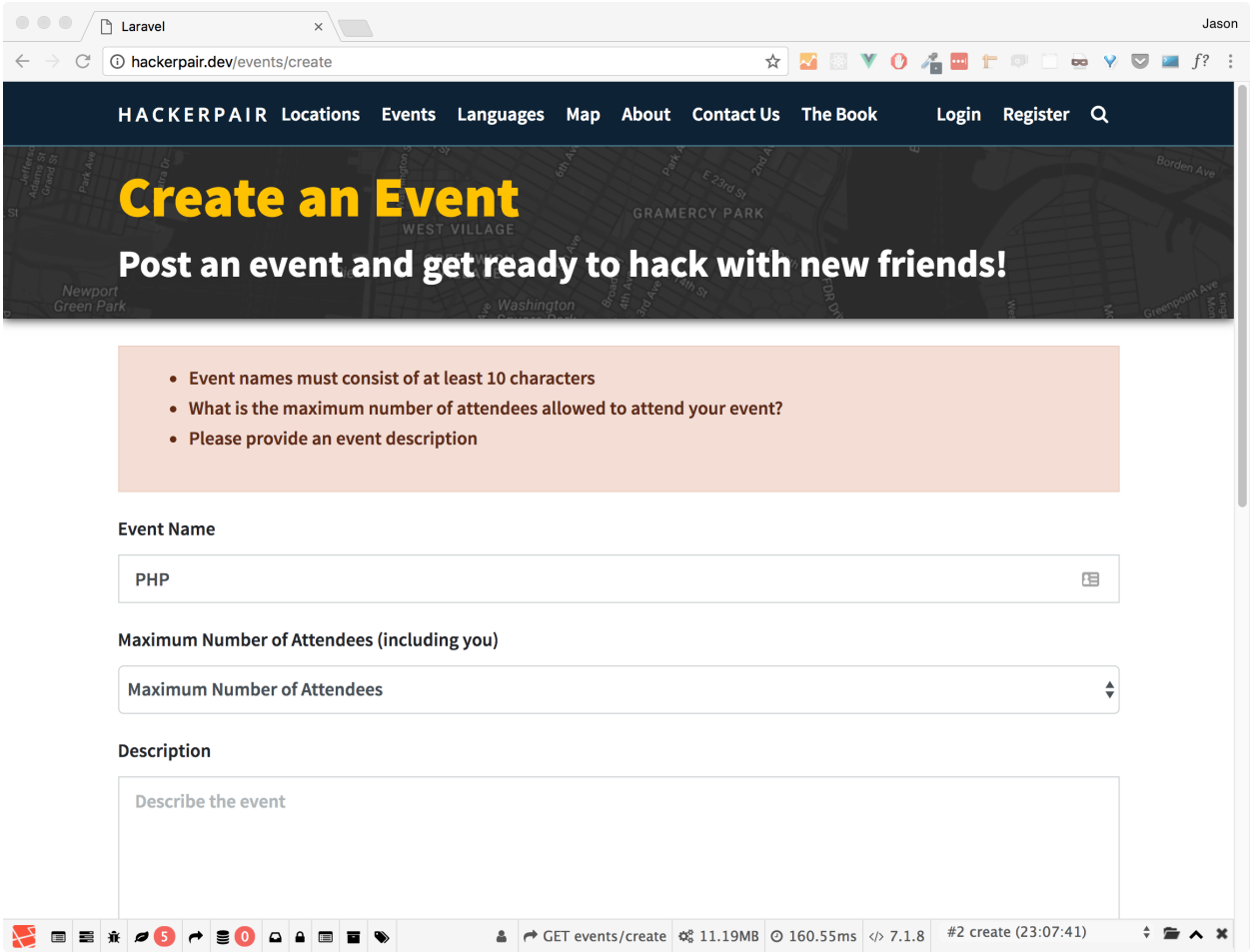
```
Please provide an event max_attendees
```

To fix this issue the next array element overrides the `require` message for the `max_attendees` field:

```
'max_attendees.required' => 'What is the maximum number of attendees
                             allowed to attend your event?'
```

This array along with the `$rules` array are passed into the `Validator::make` method along with the request input. Take special note of the trailing `validate` method. This ensures the user is redirected back to the form should a validation error occur. Omitting this method call will cause Laravel to assume you have other plans in mind should validation fail since you've overridden the default validator behavior.

With the custom messages in place, you'll see error output similar to the following should you provide an exceedingly short event name, neglect to identify the maximum number of attendees, and leave the description field blank.



Custom error messages

Laravel validators are undoubtedly powerful, and give you the utmost flexibility in terms of detecting and responding to input errors. However, the last example in particular shows that a great deal of customization can cause the method's primary purpose to get lost in the shuffle. Ideally, your controller methods will remain laser focused on performing a single task. In the case of `store`, that task is inserting a resource into the database. That's it! Many developers would argue the `store` method shouldn't be bothered with *any* other matter, including validating input! The Laravel developers agree with this sentiment, and so introduced in Laravel 5 a new way to completely encapsulate error handling logic within a separate class which can be injected into the desired controller action. I'll introduce this feature next.

Introducing Form Requests

Form requests behave in a fashion practical identical to validators, the key difference being form requests allow you to manage the validation logic separately from the action. To create a form request, navigate to your project's root directory and run the following Artisan command:

```
$ php artisan make:request EventStoreRequest
Request created successfully.
```

You're free to call the request anything you please. I tend to use compound names combining the model and action for which the request will be used. You'll find a new file named `EventStoreRequest.php` in the directory `app/Http/Requests`. The file (comments removed) looks like this:

```
<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class EventCreateRequest extends FormRequest
{

    public function authorize()
    {
        return false;
    }

    public function rules()
    {
        return [
            //
```

```
        ];
    }

}
```

As you probably surmised, the `rules` method is responsible for defining the rules associated with this request. Update the `rules` method to look like this:

```
public function rules()
{
    return [
        'name'          => 'required|min:10|max:50',
        'max_attendees' => 'required|integer|digits_between:2,5',
        'description'   => 'required'
    ];
}
```

The `authorize` method is used to determine whether the requesting client is allowed to complete the task for which this form request has been applied. We're not quite ready to deal with authorization just yet (the topic is addressed in chapter 9), so for the moment update the method to return `true` instead of `false`.

Save the changes and return to the `Events` controller. Remove all of the validation-specific logic from the `store` action, and change the action's `$request` type hint to reference `EventStoreRequest` rather than `Request`:

```
use App\Http\Requests\EventStoreRequest;

...

public function store(EventStoreRequest $request)
{

    $event = Event::create(
        $request->input()
    );

    flash('Event created!')->success();
    return redirect()->route('events.show', ['event' => $event]);

}
```

After saving these changes, return to the browser and submit the event creation form anew with one or more errors included. You should see error output similar to that found in earlier examples!

Customizing Form Request Messages

It's possible to customize form request error messages just as we did with the validators. All you need to do is add a method named `messages` to your form request which returns an array containing the desired field/validator and message mappings:

```
public function messages()
{

    return [
        'required' => 'Please provide an event :attribute',
        'max_attendees.required' => 'What is the maximum number of
         attendees allowed to attend your event?',
        'name.min' => 'Event names must consist of at least 10 characters',
        'name.max' => 'Event names cannot be longer than 50 characters',
        'max_attendees.digits_between' => 'We try to keep events cozy,
         consisting of between 2 and 5 attendees, including you.'
    ];

}
```

After saving the changes, your custom messages will be displayed whenever an associated validation error arises!

Chapter 7. Creating and Managing Model Relationships

Thus far we've been taking a fairly simplistic view of the project database, interacting with a single model (`Event`) and its underlying `events` table. However, in the real world an application's database tables are like an interconnected archipelago, with bridges connecting the islands. These allegorical causeways allow the developer to efficiently determine for instance which users have favorited a particular event, which user created an event, and find all events occurring in the state of Ohio.

Such relationships are possible thanks to a process known as *database normalization*[65]. Database normalization formally structures table relations, eliminates redundancy, and improves maintainability. Laravel works in conjunction with a normalized database to provide powerful features useful for building and traversing relations with incredible dexterity. In this chapter I'll introduce you to these wonderful capabilities.

Introducing Relations

The HackerPair application is largely based around four data entities: events, categories, users, and geographical locations. Each event is mapped to a single category, meaning each category (e.g. PHP) can be associated with multiple events. Each user is associated with a U.S. state, meaning we can determine for instance how many users live in each state. These relationships are formally defined within your project models, and fall into one of the following categories:

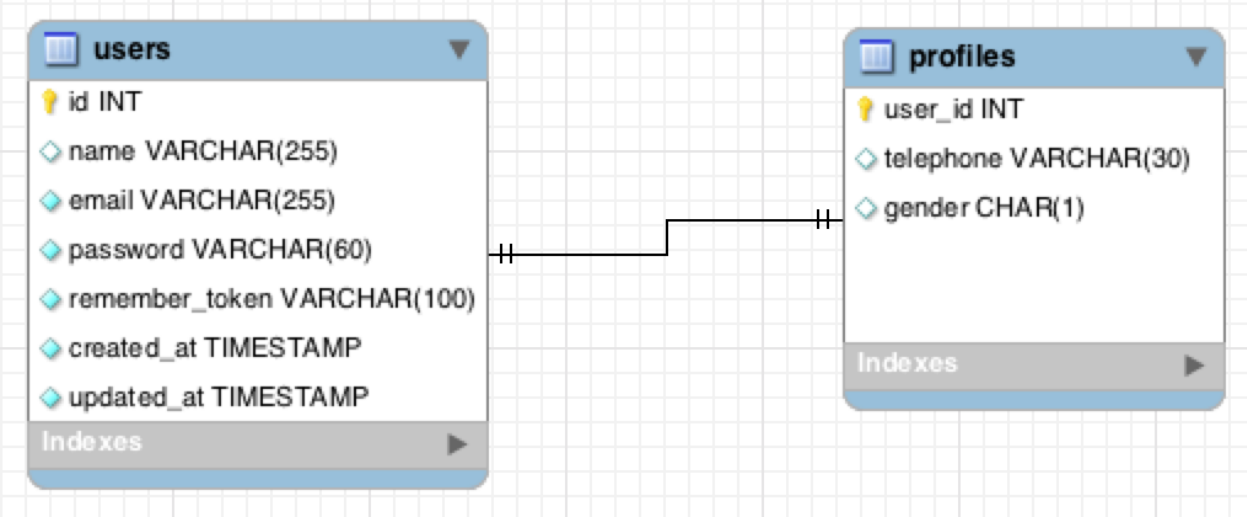
- **One-to-One Relation**: One-to-One relations are used when one entity can only belong with one other entity. For instance for organizational reasons you might choose to separate user authentication information (e-mail address and password) from profile-related characteristics such as the name, bio and phone number. Because one user can be associated with only one profile, and one profile can be associated with only one user, this would be an ideal one-to-one relation. In many cases one-to-one relations come about for purely organizational purposes, because there is often little reason to manage uniquely-related data in multiple tables.
- **Belongs To Relation**: The Belongs To relation assigns ownership of a particular record to another record. For instance, an event *belongs to* a state by configuring the database in such a way that each event record points to the state record for which it belongs. This is an important concept that often trips up beginners: always remember that the record doing the pointing is identified as *belonging to* the pointed-at record.

[65] http://en.wikipedia.org/wiki/Database_normalization

- **One-to-Many Relation**: One-to-Many relations are used when one entity can be associated with multiple entities. For instance, a state *has many* events, and each event *belongs to* a state.
- **Many-to-Many Relation**: Many-to-Many relations are used when one record can be related to multiple other records, and vice versa. For instance, if we were to expand HackerPair to include an online store selling products, the products could be associated to many categories, and each category could be associated with many products.
- **Has Many Through Relation**: The Has Many Through relation is useful when you want to interact with a table through an intermediate relation. Suppose HackerPair expanded globally and began associating users with a country, and you wanted to display the number of favorited events according to country. But the user's country ID is stored in the users table, meaning it's not possible to know whether a particular favorited event is associated with a user living in Japan without also examining the user record. You can simplify the process used to perform this sort of analysis using the Has Many Through relation.
- **Polymorphic Relation**: Polymorphic relations are useful when you want a *model* (as opposed to a table record) to belong to more than one other model. Perhaps the most illustrative example of polymorphic relation's utility involves wishing to associate comments with several different types of data (blog entries and products, for instance). It would be inefficient to create separate comment-specific models/tables for these different types of data, and so you can instead use a polymorphic relation to relate a *single* comment model to as many other models as you please without sacrificing capabilities. If you're not familiar with polymorphic relations then I'd imagine this sounds a bit like magic however I promise it will soon all make sense.

Introducing One-to-One Relations

One-to-one relationships link one row in a database table to one (and only one) row in another table. In my opinion there are generally few uses for a one-to-one relationship because the very nature of the relationship indicates the data could be consolidated within a single table. However, for the sake of demonstration let's suppose your application offered user authentication and profile management, and you wanted to separate the authentication credentials (e-mail address and password) from their profiles (name, phone number, gender). This relationship is depicted in the below diagram.



An example one-to-one relationship

To manage this relationship in Laravel you'll associate the `User` model (created automatically with every new Laravel project; I'll formally introduce this model in Chapter 7) with the model responsible for managing the profiles, which we'll call `Profile`. To create the model you can use the Artisan generator:

```
$ php artisan make:model Profile -m
```

You'll find the newly generated model inside `app/Profile.php`:

```
namespace App;

use Illuminate\Database\Eloquent\Model;

class Profile extends Model {

    //

}
```

Laravel will also generate a migration for the model's corresponding database table (`profiles`). Open up the newly created migration (inside `database/migrations`) and modify the `up` method to look like this:

```
public function up()
{
  Schema::create('profiles', function(Blueprint $table)
  {
    $table->increments('id');
    $table->integer('user_id')->unsigned();
    $table->foreign('user_id')->references('id')->on('users');
    $table->string('url');
    $table->string('telephone');
    $table->timestamps();
  });
}
```

The bolded lines are the only four you'll need to add, as the other two will already be in place when you open the file. The first line results in the addition of an integer-based column named `user_id`. The second line identifies this column as being a foreign key which references the `users` table's `id` column.

After saving the changes run the following command to create the table:

```
$ php artisan migrate
Migrated: 2017_12_22_015236_create_profiles_table
```

With the tables in place it's time to formally define the relations within the Laravel application.

Defining the One-to-One Relationship

You'll define a one-to-one relation by creating a public method typically having the same name as the related model. The method will return the value of the `hasOne` method, as demonstrated below:

```
class User extends Model {

  public function profile()
  {
      return $this->hasOne('App\Profile');
  }

}
```

Once defined, you can retrieve a user's profile information by calling the user's `profile` method. Because the relations can be chained, you could for instance retrieve a user's telephone number like this:

```
$user = User::find(1)->profile->telephone;
```

To retrieve the telephone number, Laravel will look for a foreign key in the `profiles` table named `user_id`, matching the ID stored in that column with the user's ID.

The above example demonstrates how to traverse a relation, but how is a relation created in the first place? I'll show you how to do this next.

Creating a One-to-One Relationship

You can easily create a One-to-One relation by creating the child object and then saving it through the parent object, as demonstrated in the below example:

```
$profile = new Profile;
$profile->telephone = '614-867-5309';

$user = User::find(1);
$user->profile()->save($profile);
```

Once executed, you'll find a newly created profile in the `profiles` table with a `user_id` of `1`.

Deleting a One-to-One Relation

Because a profile should not exist without a corresponding user, you'll just delete the associated profile record in the case you want to end the relationship:

```
$user = User::find(212);
$user->profile()->delete();
```

Because it is unlikely you'd want a profile's user to be deleted without also deleting the profile, since it would result in an *orphaned* profile (a profile record which points to a nonexistent user ID). One way to avoid this is by deleting the related profile record before deleting the user record, but chances are this two step process will eventually be neglected, resulting in orphaned records. Instead, you'll probably want to automate the task by taking advantage of the underlying database's ability to delete child tables when the parent table is deleted. You can specify this requirement when defining the foreign key in your table migration. I've modified the relevant lines of the earlier migration used to create the `profiles` table, attaching the `onDelete` option to the foreign key:

```
$table->integer('user_id')->unsigned();
$table->foreign('user_id')->references('id')
      ->on('users')->onDelete('cascade');
```

With the cascading delete option in place, deleting a user from the database will automatically result in the deletion of the user's corresponding profile.

Introducing the Belongs To Relationship

With the above example's `hasOne` relation in mind, it's possible to retrieve a profile attribute via a user, such as a phone number, but *not* possible to retrieve a user by way of the associated profile. This is because the `hasOne` relation is a one-way definition. You can make the relation bidirectional by defining a `belongsTo` relation in the `Profile` model, as demonstrated here:

```
class Profile extends Model {

    public function user()
    {
      return $this->belongsTo('App\User');
    }

}
```

Because the `profiles` table contains a foreign key representing the user (via the `user_id` column), each record found in `profiles` "belongs to" a record found in the `users` table. Once defined, you could retrieve a profile's associated user e-mail address based on the profile's telephone number like so:

```
$email = Profile::where('telephone', '614-867-5309')
          ->first()->user->email;
```

The Belongs To relationship certainly isn't limited to use in conjunction with One-to-One relations, and in fact you'll rarely use it in this fashion although at this point in the chapter demonstrating it in this capacity seems warranted. It will become quite clear as this chapter progresses just how indispensable Belongs To is when used in conjunction with other relations such as Has Many.

Introducing One-to-Many (Has Many) Relationships

The One-to-Many (also known as the Has Many) relationship is useful when you want to relate one table record to one or many other table records. The Has Many relation is used extensively

throughout the HackerPair application, so in this section we'll look at some actual code. To recap from the chapter introduction, the One-to-Many relation is used when you want to relate a single table record to multiple table records. For instance a U.S. state can be associated with multiple events, meaning we'll need to relate the `Event` and `State` models using a One-to-Many relation.

Creating the State Model

Earlier in the book we created the `Event` model, meaning we'll need to create the `State` model in order to begin associating events with states. Use Artisan to generate the `State` model and associated migration:

```
$ php artisan make:model State --migration
```

You'll find the newly generated model inside `app/State.php`:

```
use Illuminate\Database\Eloquent\Model;

class State extends Model {
    //
}
```

Open the newly created corresponding migration file (found in `database/migrations`) and modify the `up()` method to look like this:

```
public function up()
{
  Schema::create('states', function(Blueprint $table)
  {
    $table->increments('id');
    $table->string('name');
    $table->string('abbreviation');
    $table->timestamps();
  });
}

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

After saving the changes, run the migration:

```
$ php artisan migrate
Migrated: 2017_12_22_005821_create_states_table
```

Next you'll want to update the `events` table to store a foreign key pointing to the newly created `states` table:

```
$ php artisan make:migration add_state_id_to_events_table --table=events
Created migration: 2017_12_22_add_state_id_to_events_table
```

Open the newly created migration and modify the `up` and `down` methods to look like this:

```
public function up()
{
    Schema::table('events', function (Blueprint $table) {
        $table->integer('state_id')->unsigned()->nullable();
        $table->foreign('state_id')
            ->references('id')->on('states');
    });
}

public function down()
{
    Schema::table('events', function (Blueprint $table) {
        $table->dropForeign('events_state_id_foreign');
        $table->dropColumn('state_id');
    });
}
```

After saving the changes, run your migrations again to add the `state_id` field to the `events` table.

Defining the One-to-Many Relation

With the `State` model and underlying `states` table created, and the `state_id` field added to the `events` table, it's time to create the relation. Open the `State` model and create the following `events` method:

```
class State extends Model {

  ...

  public function events() {
      return $this->hasMany('App\Event');
  }

}
```

You'll also want to define the opposite side of the relation within the `Event` model using the `belongsTo` method:

```
class Event extends Model {

  ...

  public function state()
  {
    return $this->belongsTo('App\State');
  }

}
```

With the relation defined, let's next review how to associate an event with a state.

Associating an Event with a State

Laravel supports a few different ways to associate data related through a Has Many relationship. The easiest involves assigning the foreign key ID to the record. For instance, suppose the state of Ohio is associated with ID 41. You could relate an event to Ohio like so:

```
$event = new Event;
$event->name = "Laravel Hacking and Pizza";
$event->state_id = 41;
$event->save();
```

This approach is pretty standard when the ID is passed along via form submission. For instance, the HackerPair event creation form includes a select box containing all of the state names and their IDs. It is generated like this:

```
<div class="form-group">
  {!! Form::label('state_id', "State", ['class' => 'control-label']) !!}
  {!! Form::select('state_id',
  \App\State::orderBy('name', 'asc')->pluck('name', 'id'), null,
  ['class' => 'form-control input-lg']);
  !!}
</div>
```

That `state_id` form field is passed along with the others and can be mass-assigned provided the `state_id` field is included in the `Event` model's `$fillable` array:

```
$event = Event::create(
    $request->input()
);
```

If the `state_id` field isn't included in `$fillable`, then you could assign it just as we did in the previous example. But what if you didn't know the state ID, or wanted to first confirm the ID does in fact belong to a record in the `states` table? You could assign the state to the event like so:

```
$state = State:where('name', 'Ohio')->first();
$event = new Event;
$event->name = "Laravel Hacking and Pizza";
$event->state = $state;
$event->save();
```

Iterating Over a State's Events

The HackerPair application gives users the ability to view each state's associated events. For instance, go to `http://hackerpair.com/states/oh` and you'll see a list of events associated with Ohio. This route is handled by the `States` controller's `show` method:

```
public function show(State $state)
{

    $events = $state->events()->paginate();

    return view('states.show')
      ->withState($state)
      ->withEvents($events);

}
```

The `$events` variable contains a collection of `Event` objects, meaning you can iterate over it in the view just like you would any other collection.

Sorting and Filtering Related Records

You'll often wish to retrieve a filtered collection of related records. For instance the user might desire to only see completed list tasks. You can do so by filtering on the `tasks` table's `done` column:

```
$publishedEventsInOhio = State::find(41)
                          ->events()
                          ->where('published', true)
                          ->get();
```

If you plan on always filtering events in this fashion you can clean up the code a bit by modifying the `State` model's `events` method:

```
public function events()
{
  return $this->hasMany('App\Event')->where('published', true);
}
```

Modifying a relationship's default behavior is also useful for sorting. For instance, you can use `orderBy` to determine the order in which a category's tasks are returned:

```
public function events()
{
  return $this->hasMany('App\Event')
    ->orderBy('created_at', 'desc')->where('published', true);
}
```

Introducing Many-to-Many Relations

You'll use the many-to-many relation when the need arises to relate a record in one table to one or several records in another table, and vice versa. For instance, a HackerPair user can attend many events, and an event can be associated with many users. In this section you'll learn how to create the intermediary table used to manage the relation (known as a *pivot table*), define the relation within the respective models, and manage the relation data.

Creating the Pivot Table

Many-to-many relations require an intermediary table to manage the relation. The simplest implementation of the intermediary table, known as a *pivot table*, would consist of just two columns for storing the foreign keys pointing to each related pair of records. The pivot table name should be formed by concatenating the two related model names together with an underscore separating the names. Further, the names should appear in alphabetical order. Therefore if we were creating a many-to-many relationship between the `Event` and `User` models, the pivot table name would be `event_user`:

```
$ php artisan make:migration create_event_user_table --create=event_user
Created Migration: 2017_12_22_011409_create_event_user_table
```

Next, open up the newly created migration and modify the `up` method to look like this:

```
public function up()
{
  Schema::create('event_user', function(Blueprint $table)
  {
      $table->integer('event_id')->unsigned()->nullable();
      $table->foreign('event_id')->references('id')
            ->on('events')->onDelete('cascade');

      $table->integer('user_id')->unsigned()->nullable();
      $table->foreign('user_id')->references('id')
            ->on('users')->onDelete('cascade');

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

This syntax is no different than that used for the earlier `events` table migration. The only real difference is that we're referencing two foreign keys rather than one.

After saving the changes run Artisan's `migrate` command to create the table:

```
$ php artisan migrate
```

Defining the Many-to-Many Relation

With the tables in place it's time to define the many-to-many relation within the respective models. Open the `Event` model and add the following method to the class:

```
public function users()
{
    return $this->belongsToMany('App\User')
      ->withTimestamps();
}
```

Notice I've chained the `withTimestamps` method to the return statement. This instructs Laravel to additionally update the `event_user` table timestamps when saving a new record. If you choose to omit the `created_at` and `updated_at` timestamps from this pivot table (done by removing `$table->timestamps` from the migration), you can omit the `withTimestamps` method).

Save the changes and then open the `User` model, adding the following method to the class:

```
public function events()
{
    return $this->belongsToMany('App\Event')
      ->withTimestamps();
}
```

After saving these changes you're ready to begin using the relation!

Associating Records Using the Many-to-Many Relation

You can associate records using the many-to-many relation in the same way as was demonstrated for one-to-many relations; just traverse the relation and use the `save` method, as demonstrated here:

```
$event = Event::find(1);

$user = User::find(42);

$event->users()->save($user);
```

After executing this code you'll see an association between this event and user has been created in the `event_user` table:

```
mysql> select * from event_user;
+-----------+---------+------------+------------+
| event_id  | user_id | created_at | updated_at |
+-----------+---------+------------+------------+
|         1 |      42 | 2017...    | 2017...    |
+-----------+---------+------------+------------+
```

You can alternatively use the `attach` and `detach` methods to associate and disassociate related records. For instance to both associate and immediately persist a new relationship between an event and user, you can pass the `Category` object or its primary key into `attach`. Both variations are demonstrated here:

```
$event = Event::find(2);

$user = User::find(42);

// In this example we're passing in a User object
$event->users()->attach($user);

// The number 5 is a user's primary key
$event->users()->attach(42);
```

You can also pass an array of IDs into `attach`:

```
$event->users()->attach([39,42]);
```

To disassociate a user from an event, you can use `detach`, passing along either the `User` object, an object's primary key, or an array of primary keys:

```
// Pass the User object into the detach method
$event->users()->detach(User::find(42));

// Pass a user's ID
$event->users()->detach(3);

// Pass along an array of user IDs
$event->users()->detach([39,42]);
```

Determining if a Relation Already Exists

Laravel will not prevent you from duplicating an association, meaning the following code will result in a list being associated with the same category twice:

```
$event = Event::find(2);

$user = User::find(42);

$event->users()->save($user);
$event->users()->save($user);
```

If you have a look at the `event_user` table you'll see that the `User` record associated with the primary key `42` has been twice related to the `Event` record associated with the primary key `2`:

```
mysql> select * from event_user;
+-----------+---------+------------+------------+
| event_id  | user_id | created_at | updated_at |
+-----------+---------+------------+------------+
|         2 |      42 | 2017...    | 2017...    |
|         2 |      42 | 2017...    | 2017...    |
+-----------+---------+------------+------------+
```

You can avoid this by first determining whether the relation already exists using the `contains` method:

```
$event = Event::find(2);

$user = User::find(42);

if ($event->users->contains($user))
{

  return redirect()->route('events.show', [$event->id])
    ->with('message', 'Duplicate entry!');

} else {

  $event->users()->save($event);

  return redirect()->route('events.show', [$event->id])
    ->with('message', 'The user has joined the event!');

}
```

Saving Multiple Relations Simultaneously

You can use the `saveMany` method to save multiple relations at the same time:

```
$user = User::find(1);

$events = Event::findMany([15, 19, 25]);

$user->events()->saveMany($events);
```

Traversing the Many-to-Many Relation

You'll traverse a many-to-many relation in the same fashion as described for the one-to-many relation; just iterate over the collection:

```
$event = Event::find(2);

...

<ul>
  @forelse($event->users as $user)
    <li>{{ $user->email }}</li>
  @empty
    <li>Nobody has signed up to attend this event</li>
  @endforelse
</ul>
```

Because the relation is defined on each side, you're not limited to traversing an event's attendees! You can also traverse a user's events:

```
$user = User::find(2);

...

<ul>
  @forelse($user->events as $event)
    <li>{{ $event->name }}</li>
  @empty
    <li>This user is not attending any events!</li>
  @endforelse
</ul>
```

Synchronizing Many-to-Many Relations

Suppose you provide add a feature that allows users to easily manage attendance status for multiple events. Because the user can both select and deselect events, you must take care to ensure that not only are the selected events associated with the user, but also that any *deselected* events are disassociated with the user. The implementation of this feature may seem rather daunting. Fortunately, Laravel offers a method named `sync` which you can use to synchronize an array of primary keys with those already found in the database. For instance, suppose events associated with the IDs `7`, `12`, `52`, and `77` were passed into the action where you'd like to synchronize the user's events. You can pass the IDs into `sync` as an array like this:

```
$events = [7, 12, 52, 77];

$user = User::find(42);

$user->events()->sync($events);
```

Once executed, the `User` record identified by the primary key `42` will be associated *only* with the events identified by the primary keys `7`, `12`, `52`, and `77`, even if prior to execution the `User` record was additionally associated with other events.

Managing Additional Many-to-Many Attributes

Thus far the many-to-many examples presented in this chapter have been concerned with a join table consisting of two foreign keys and optionally the `created_at` and `updated_at` timestamps. But what if you wanted to manage additional attributes within this table, such as a comment an attendee could optionally include when signing up to attend an event?

Believe it or not adding other attributes is as simple as including them in the table schema. For instance let's create a migration that adds a column named `comment` to the `event_user` table created earlier in this section:

```
$ php artisan make:migration add_comment_to_event_user_table
Created Migration: 2017_12_23_012822_add_descripti...
```

Next, open up the newly generated migration file and modify the `up()` and `down()` methods to look like this:

```
public function up()
{
  Schema::table('event_user', function($table)
  {
      $table->string('comment');
  });
}

public function down()
{
  Schema::table('event_user', function($table)
  {
      $table->dropColumn('comment');
  });
}
```

Save the changes and be sure to migrate the change into the database:

```
$ php artisan migrate
Created Migration: 2016_12_23_012822_add_comment...
```

Finally, you'll need to modify the `Event` and `User` model's methods to identify the additional pivot column using the `withPivot()` method. Here's the revised `User` model method:

```
public function events()
{
    return $this->belongsToMany('App\User')
      ->withPivot('comment')
      ->withTimestamps();
}
```

And here is the revised `Event` model method:

```
public function users()
{
    return $this->belongsToMany('App\Event')
      ->withPivot('comment')
      ->withTimestamps();
}
```

With the additional column and relationship tweak in place all you'll need to do is adjust the syntax used to relate events with the user. You'll pass along the event ID along with the comment key and desired value, as demonstrated here:

```
$user = User::find(42);
$user->events()->attach(
  [12 => ['comment' => 'I am excited to attend this event!']]
);
```

If you later wished to update an attribute associated with an existing record, you can use the `updateExistingPivot` method, passing along the event's foreign key along with an array containing the attribute you'd like to update along with its new value:

```
$user->events()->updateExistingPivot(12,
  ['comment' => 'This event is going to be great!']
);
```

Introducing Has Many Through Relations

Suppose HackerPair's CEO has just returned from the "Mo Big Data Mo Money" conference, flush with ideas regarding how user data can be exploited and sold to the highest bidder. He's asked you to create a new feature that summarizes the numbers of events favorited according to state. The `User` model already includes a `state_id` which is used to identify each user's state-of-residence, so you can tally up users according to state. This means the user/state relationship would look like this:

```
class User extends Model {

  public function state()
  {
    return $this->belongsTo('App\State');
  }

}

class State extends Model {

  public function users()
  {
      return $this->hasMany('App\User');
  }

}
```

Because the `users` table contains the foreign key reference to the `states` table's ID, and not the user's favorited events, how can you relate favorited events with states? The SQL query used to mine this sort of data is pretty elementary:

```
SELECT count(favorite_events.id), states.name FROM favorite_events
  LEFT JOIN users on users.id = favorited_events.user_id
  LEFT JOIN states ON states.id = users.state_id
  GROUP BY states.name;
```

But how might you implement such a feature within your Laravel application? Enter the Has Many Through relation. The Has Many Through relation allows you to create a shortcut for querying data available through distantly related tables. Just add the following relation to the `State` model:

```
public function favorites()
{
    return $this->hasManyThrough('App\FavoriteEvent', 'App\User');
}
```

This relation gives the `State` model the ability to access the `FavoriteEvent` model *through* the `User` model. After saving the model, you'll be able to for instance iterate over all favorited events created by users residing in Ohio:

```
$state = State::where('name', 'Ohio')->first();

...

<ul>
  @foreach($state->favorites as $favorite) {
    <li>{{$favorite->name}}</li>
  @endforeach
</ul>
```

Introducing Polymorphic Relations

When considering an interface for commenting on different types of application data (products and blog posts, for example), one might presume it is necessary to manage each type of comment separately. This approach would however be repetitive because each comment model would presumably consist of the same data structure. You can eliminate this repetition using a *polymorphic relation*, resulting in all comments being managed via a single model.

Let's work through an example that would use polymorphic relations to add commenting capabilities to the `User` and `Event` models. Begin by creating a new model named `Comment`:

```
$ php artisan make:model Comment --migration
```

You'll find the newly generated model inside `app/Comment.php`:

```
use Illuminate\Database\Eloquent\Model;

class Comment extends Model {

    //

}
```

Next, open up the newly generated migration file and modify the `up()` method to look like this:

```
Schema::create('comments', function(Blueprint $table)
{
  $table->increments('id');
  $table->text('body');
  $table->integer('commentable_id');
  $table->string('commentable_type');
  $table->timestamps();
});
```

Finally, save the changes and run the migration:

```
$ php artisan migrate
Migrated: 2017_12_23_223902_create_comments_table
```

Because the `Comment` model serves as a central repository for comments associated with multiple different models, we require a means for knowing both which model and which record ID is associated with a particular comment. The `commentable_type` and `commentable_id` fields serve this purpose. For instance, if a comment is associated with an event, and the event record associated with the comment has a primary key of `453`, then the comment's `commentable_type` field will be set to `Event` and the `commentable_id` to `453`.

Logically you'll want to attach other fields to the `comments` table if you plan on for instance assigning ownership to comments via the `User` model, or would like to include a title for each comment.

Next, open the `Comment` model and add the following method:

```
class Comment extends Model {

    public function commentable()
    {
      return $this->morphTo();
    }

}
```

The `morphTo` method defines a polymorphic relationship. Whenever you read `morphTo` just think "belongs To" but for polymorphic relationships, since the record will belong to whatever model is defined in the `commentable_type` field. This defines just one side of the relationship; you'll also want to define the inverse relation within any model that will be commentable, creating a method that determines which model is used to maintain the comments, *and* referencing the name of the method used in the polymorphic model:

```
class Event extends Model {

  public function comments()
  {
    return $this->morphMany('App\Comment', 'commentable');
  }

}
```

With these two methods in place, it's time to begin using the polymorphic relation! The syntax for adding, removing and retrieving comments is straightforward; in the following example we'll attach a new comment to a list:

```
$event = Event::find(1);

$c = new Comment();

$c->body = 'Great event!';

$event->comments()->save($c);
```

After saving the comment, review the database and you'll see a record that looks like the following:

```
mysql> select * from events;
+----+--------------+----------------+------------------+------------+...
| id | body         | commentable_id | commentable_type | created_at |...
+----+--------------+----------------+------------------+------------+...
|  1 | Great event! |              1 | App\Event        | 2017-...   |...
+----+--------------+----------------+------------------+------------+...
```

The event's comments are just a collection, so you can easily iterate over it. You'll retrieve the event within the controller per usual:

```
public function show()
{
  $event = Event::find(1);
  return view('events.show')->with('event', $event);
}
```

In the corresponding view you'll iterate over the comments collection:

```
  @foreach ($event->comments as $comment)
    <p>
      {{ $comment->body }}
    </p>
  @endforeach
```

To delete a comment you can of course just delete the comment using its primary key.

Eager Loading

When your project first launches you probably don't have to worry too much about query optimization. However performance could eventually begin to suffer once your database becomes sufficiently populated should your application need to perform multiple and repeated table joins when displaying data.

Consider for instance the HackerPair home page (http://hackerpair.com). It displays a paginated list of the most recently created events. Each event includes the event name, the host's name, the category, city and state, start date and time, and the attendee count. Thanks to Laravel's wonderful object-oriented syntax, it would be tempting to just use the following query:

```
$events = Event::orderBy('start_date', 'desc')->paginate();
```

This approach can be problematic though, because when the view is rendered Laravel will need to perform multiple additional queries to retrieve the related data. In fact, the Laravel Debugbar indicates a total of 30 queries are executed to render the first page of paginated events!

If you're going to retrieve a record's related data, you can dramatically reduce the number of queries by "eager loading" the data using the `with` method:

```
1  $events = Event::with(['category', 'organizer', 'state'])
2              ->orderBy('start_date', 'desc')
3              ->paginate();
```

After updating the query to use `with`, the total number of queries required to load the first page of paginated events on the HackerPair home page has been reduced from 30 to 12!

There you have it, eager loading demystified! Be sure to incorporate this approach into your applications, I guarantee you'll see significant performance improvements as a result of making this seemingly minor change.

Conclusion

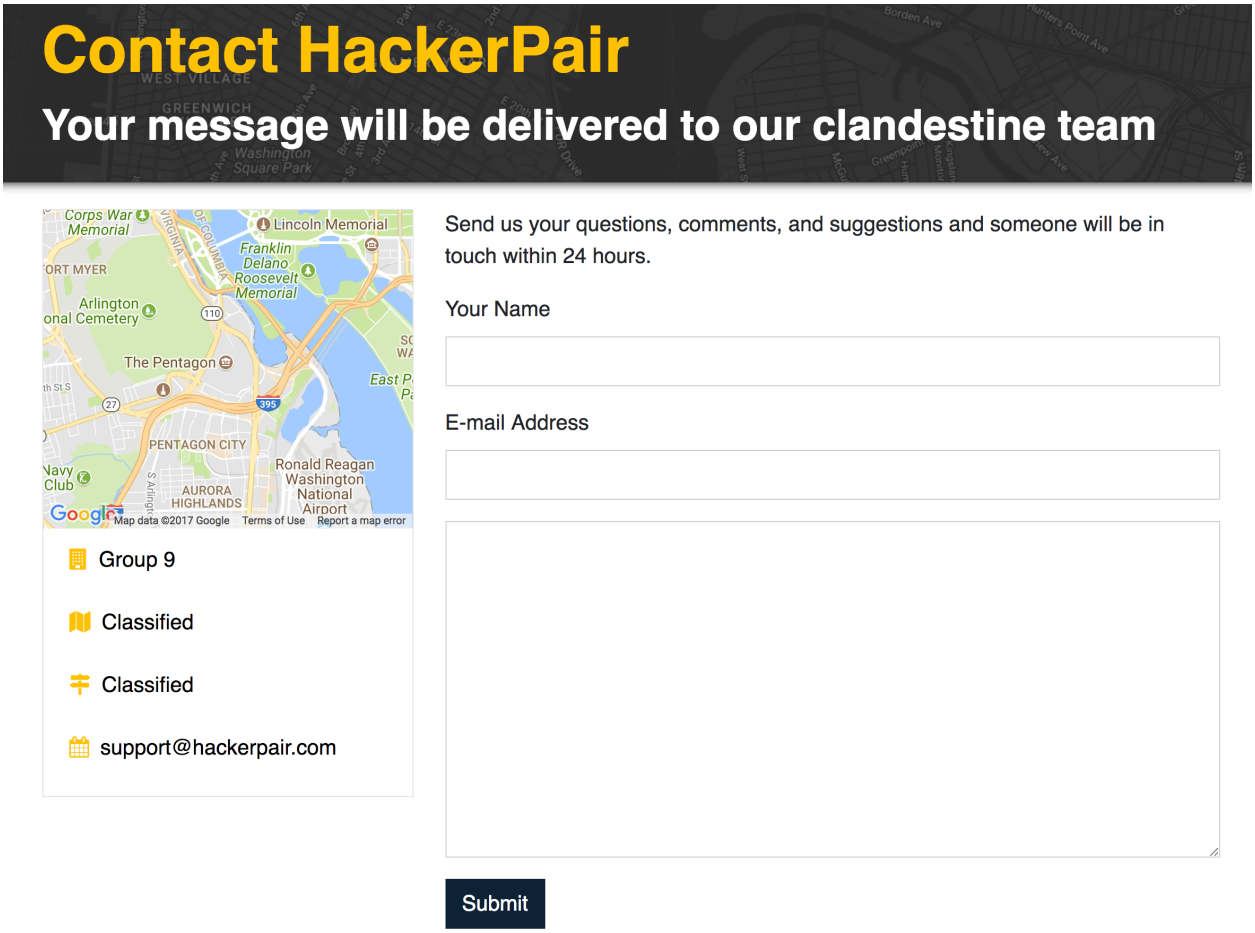
I'd imagine this to be the most difficult chapter in the book, primarily because you not only have to understand the syntax used to manage and traverse relations but also be able to visualize at a conceptual level the different ways in which your project data should be structured. Although it's a tall order, once you do have a solid grasp on the topics presented in this chapter there really will be no limit in terms of your ability to build complex database-driven Laravel projects! As always if you don't understand any topic discussed in this chapter, or would like to offer some input regarding how any of the material can be improved, be sure to e-mail me at wj@wjgilmore.com.

Chapter 8. Sending E-mails

No matter how much attention is given to text messages, mobile notifications, and third-party platforms such as Slack, e-mail remains the undisputed king of internet-based communication. That said, no matter what type of application you plan on building, chances are you'll want to automate the generation and delivery of e-mail to your users (not to mention yourself).

Fortunately the Laravel developers are cognizant of this common need, and have incorporated an incredibly convenient and well-organized solution for managing application-based e-mails. In this chapter you'll learn all about constructing, sending, and testing e-mails inside your Laravel application.

Let's kick things off by building a simple contact form consisting of three fields, including the user's name, email address, and message (see below figure).



HackerPair's contact form

The contact form presentation and processor will require just two controller methods, meaning you could get away with managing them inside a general administrative controller. However I always prefer to follow RESTful conventions whenever possible, and so suggest instead creating a dedicated controller for this purpose. In the case of HackerPair this feature is managed by the `Contact` controller's `create` and `store` actions (`create` presents the form via the `GET` method and `store` processes it via `POST`). Because we're going to use only two of the seven REST methods you can eliminate the need to delete unused methods by creating a "plain" controller:

```
$ php artisan make:controller ContactController
Controller created successfully.
```

Next, to route users to the contact form using the convenient `/contact` shortcut you'll need to define two aliases in the `routes/web.php` file:

```
Route::get('contact', 'ContactController@create')->name('contact.create');
Route::post('contact', 'ContactController@store')->name('contact.store');
```

Next, you'll need to add the `create` and `store` actions to the newly created `Contact` controller, because we didn't specify the `--resource` option when generating the controller. Modify the controller to look like this:

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;

use App\Http\Requests;

class ContactController extends Controller {

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

    public function store()
    {
    }

}
```

The `create` action has been configured to serve a view named `create.blade.php` found in the directory `resources/views/contact`. However we haven't yet created this particular view so let's do so next.

Creating the Contact Form

Earlier in this chapter I showed you a screenshot of the *rendered* form HTML. Note my emphasis on *rendered* because you won't actually hand-code the form! Instead, I suggest you use a fantastic Laravel package called LaravelCollective/html[66] (introduced in chapter 2) to manage this tedious task for you. Below I've pasted in the relevant section of code found in HackerPair's contact form:

[66] https://github.com/LaravelCollective/html

```
{!! Form::open(['route' => 'contact.store']) !!}

<div class="form-group">
    {!! Form::label('name', 'Your Name') !!}
    {!! Form::text('name', null, ['class' => 'form-control']) !!}
</div>

<div class="form-group">
    {!! Form::label('email', 'E-mail Address') !!}
    {!! Form::text('email', null, ['class' => 'form-control']) !!}
</div>

<div class="form-group">
    {!! Form::textarea('msg', null, ['class' => 'form-control']) !!}
</div>

{!! Form::submit('Submit', ['class' => 'btn btn-info']) !!}

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

View the HackerPair Contact Form View Code

You can view the HackerPair contact form view on GitHub[67].

You learned how to install this package in chapter 2, but at that point we just configured the HTML Facade. To take advantage of the form-specific tags you'll need to additionally add the following alias to the `config/app.php` `aliases` array:

```
'Form'=> Collective\Html\FormFacade::class
```

If this is your first encounter with the `Form::open` helper then I'd imagine this example looks a tad scary. However once you build a few forms in this fashion I promise you'll wonder how you ever got along without it. Let's break down the key syntax used in this example:

```
{!! Form::open(['route' => 'contact.store', 'class' => 'form']) !!}
...
{!! Form::close() !!}
```

The `Form::open` and `Form::close()` methods work together to generate the form's opening and closing tags. The `Form::open` method accepts an array containing various settings such as the route

[67] https://github.com/wjgilmore/hackerpair/blob/master/resources/views/contact/create.blade.php

alias which in this case points to the `About` controller's `store` method, and a class used to stylize the form. The default method is `POST` however you can easily override the method to instead use `GET` by passing `'method' => 'get'` into the array. Additionally, the `Form::open` method will ensure the aforementioned CSRF-prevention `_token` hidden field is added to the form.

Next you'll see a series of methods used to generate the various form fields. This is a relatively simplistic form therefore only a few of the available field generation methods are used, including `Form::label` (for creating form field labels), `Form::text` (for creating form text fields), `Form::textarea` (for creating a form text area), and `Form::submit` (for creating a submit button). Note how the `Form::text` and `Form::textarea` methods all accept as their first argument a model attribute name (`name`, `email`, and `msg`, respectively). All of the methods also accept an assortment of other options, such as class names and HTML5 form attributes.

Once you add this code to your project's `resources/views/contact/create.blade.php` file, navigate to `/contact` and you should see a form similar to that found at `http://hackerpair.com/contact`!

With the form created, we'll next need to create the logic used to process the form contents and send the feedback to the site administrator via e-mail.

Creating the Contact Form Request

Laravel 5 introduced a feature known as a *form request*. This feature is intended to remove form authorization and validation logic from your controllers, instead placing this logic in a separate class. HackerPair uses form requests in conjunction with each form used throughout the site and I'm pleased to report this feature works meets its goal quite nicely.

To create a new form request you can use Artisan's `make:request` feature:

```
1 $ php artisan make:request ContactFormRequest
2 Request created successfully.
```

This created a file named `ContactFormRequest.php` that resides in the directory `app/Http/Requests/`. The class skeleton looks like this (comments removed):

```
1 namespace App\Http\Requests;
2
3 use Illuminate\Foundation\Http\FormRequest;
4
5 class ContactFormRequest extends FormRequest {
6
7   public function authorize()
8   {
9     return false;
```

```
10    }
11
12    public function rules()
13    {
14      return [
15        //
16      ];
17    }
18
19  }
```

View the HackerPair Contact Form Request Code

You can view the HackerPair contact form request on GitHub[68].

The `authorize` method determines whether the current user is authorized to interact with this form. I'll talk more about the purpose of this method in chapter 7. For the purposes of the contact form we want any visitor to submit a contact request and so modify the method to return `true` instead of `false`:

```
1  public function authorize()
2  {
3    return true;
4  }
```

The `rules` method defines the validation rules associated with the fields found in the form. The contact form has three fields, including `name`, `email`, and `msg`. All three fields are required, and the `email` field must be a syntactically valid e-mail address, so you'll want to update the `rules` method to look like this:

```
1  public function rules()
2  {
3    return [
4      'name'    => 'required',
5      'email'   => 'required|email',
6      'msg'     => 'required'
7    ];
8  }
```

[68] https://github.com/wjgilmore/hackerpair/blob/master/app/Http/Requests/ContactFormRequest.php

The `required` and `email` validators used in this example are just a few of the many supported validation features. See chapter 6 for more information about these rules. In the examples to come I'll provide additional examples demonstrating other available validators. Additionally, note how you can use multiple validators in conjunction with a form field by concatenating the validators together using a vertical bar (`|`).

After saving the changes to `ContactFormRequest.php` open the `Contact` controller and modify the `store` method to look like this:

```
use App\Http\Requests\ContactFormRequest;

...

class AboutController extends Controller {

  public function store(Request $request)
  {

    $contact = [];

    $contact['name'] = $request->get('name');
    $contact['email'] = $request->get('email');
    $contact['msg'] = $request->get('msg');

    // Mail delivery logic goes here

    flash('Your message has been sent!')->success();

    return redirect()->route('contact.create');

  }

}
```

While we haven't yet added the e-mail delivery logic, believe it or not this action is otherwise complete. This is because the `ContactForm` form request will handle the validation *and* display of validation error messages should validation fail. For instance submitting the contact form without completing any of the fields will result in three validation error found presented in the below screenshot being displayed:

Displaying contact form validation errors

These errors won't appear out of thin air of course; they'll be displayed via the `$errors` array. I typically insert the following code into my project layout so the `$errors` variable is available whenever needed:

```
@if ($errors->any())
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif
```

You'll also want to inform the user of a successful form submission. To do so you can use a flash message, which is populated in the `store` method ("Your message has been sent!"). While it's possible to do this using native Laravel code, I prefer to rely on the great laracasts/flash[69] package for doing so. Chapter 5 shows you how to integrate this package into your application.

Only one step remains before the contact form is completely operational. We'll need to configure Laravel's mail component and integrate e-mail delivery functionality into the `store` method. Let's complete these steps next.

Configuring Laravel's Mail Component

Thanks to integration with the popular SwiftMailer[70] package, it's easy to send e-mail through your Laravel application. All you'll need to do is make a few changes to the `config/mail.php` configuration file. In this file you'll find a number of configuration settings:

- `driver`: Laravel supports several mail drivers, including `SMTP`, PHP's `mail` function, the Sendmail MTA, and the Mailgun[71] and Mandrill[72] e-mail delivery services. You'll set the

[69] https://github.com/laracasts/flash/
[70] http://swiftmailer.org/
[71] http://www.mailgun.com/
[72] https://mandrill.com/

`driver` setting to the desired driver, choosing from `smtp`, `sendmail`, `mailgun`, `mandrill`, `ses`, `sparkpost`, and `array`. You could also optionally set `driver` to `log` in order to send e-mails to your development log rather than bother with actually sending them out during the development process.

- `host`: The `host` setting sets the host address of your SMTP server.
- `port`: The `port` setting sets the port used by your SMTP server.
- `from`: If you'd like all outbound application e-mails to use the same sender e-mail and name, you can set them using the `from` and `address` settings defined in this array.
- `encryption`: The `encryption` setting sets the encryption protocol used when sending e-mails.
- `username`: The `username` setting sets the account username.
- `password`: The `password` setting sets the account password.
- `sendmail`: The `sendmail` setting sets the server Sendmail path should you be using the `sendmail` driver.
- `markdown`: The `markdown` setting is used to configure your design templates if Markdown format is used within your e-mails.

Although you'll commonly find tutorials demonstrating how to use a Gmail account to send e-mail through a PHP application, I suggest against doing so. Using an Gmail account for such purposes may seem convenient because of their ubiquity, however Google has made it increasingly difficult to use Gmail accounts in this manner due largely to security and spamming concerns. Instead, consider using a third-party service created precisely for such purposes. One solution is mailgun[73], which offers a free pricing tier for users sending less than 10,000 e-mails per month.

While we all love no-cost solutions, be aware the pricing terms can change with little notice. Previous editions of this book promoted another solution which pulled the rug on their free tier after building a rather large following. I won't do them the favor of mentioning the name here, and only say you should be prepared to switch from one service to another should pricing terms suddenly become unfavorable.

To create a Mailgun account head over to https://mailgun.com/[74]. After creating your account you'll be provided with a sandbox console which you can use to experiment with sending e-mail using curl[75] or a variety of programming languages (PHP included). Don't worry about this for now because Laravel includes Mailgun support so you won't have to worry about integrating any custom PHP code to send e-mail from your application.

You'll also be prompted to identify the domain from which your Mailgun-managed e-mails will be sent. At this point in time you're not required to do so in order to begin integrating Mailgun into your application because Mailgun also creates a sandbox domain which can be used to send up to 300 messages daily. To demonstrate integration we'll use this sandbox domain so click the `Continue`

[73] https://www.mailgun.com/
[74] https://mailgun.com/
[75] https://curl.haxx.se/

`To Your Control Panel` link to continue. You'll be taken to the Mailgun dashboard where you'll find your sandbox domain. Click on the domain name and you'll be taken to a page containing your sandbox domain authentication credentials. This is where configuration can be a bit confusing, because you actually won't need to modify the `config/mail.php` settings when using a third-party service such as Mailgun; instead, you'll only need to be concerned with the following settings found in `config/services.php`:

```
'mailgun' => [
    'domain' => env('MAILGUN_DOMAIN'),
    'secret' => env('MAILGUN_SECRET'),
],
```

If you return to the Mailgun domain settings for your sandbox, you'll find both the domain (e.g. `sandbox123456.mailgun.org`) and the secret (identified as the `API KEY`). Add the following two lines to your `.env` file, updating my placeholders to reflect your actual credentials:

```
MAILGUN_DOMAIN=sandbox123456.mailgun.org
MAILGUN_SECRET=key-supersecret
```

Additionally, add the following three variables, which we'll use when sending the e-mail. Be sure to update `MAIL_FROM` to reflect the username associated with your sandbox domain settings. You can set `MAIL_TO` to the desired recipient, and `MAIL_NAME` just identifies the name associated with the sender's address:

```
MAIL_FROM=postmaster@sandbox123456.mailgun.org
MAIL_NAME="HackerPair Support"
MAIL_TO=wj@wjgilmore.com
```

After saving these changes, we'll need to install the Guzzle HTTP client, which is used to facilitate communication with web services:

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

Generating the Mailable Class

Laravel offers a great solution for managing the code responsible for generating and delivering e-mails sent from your application. It's called a Mailable, and you can generate a mailable class using Artisan:

```
$ php artisan make:mail ContactEmail
```

This will create a new skeleton class found in `app/Mail/ContactEmail.php`. Open this file and you'll find the following template:

```
<?php

namespace App\Mail;

use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;
use Illuminate\Contracts\Queue\ShouldQueue;

class ContactEmail extends Mailable
{
    use Queueable, SerializesModels;

    public function __construct()
    {
        //
    }

    public function build()
    {
        return $this->view('view.name');
    }
}
```

View the HackerPair ContactEmail Mailable Code

You can view the HackerPair contact mailable class on GitHub[76].

The constructor is responsible for initializing any data passed into the Mailable class from the caller (which as you'll soon see will be the `Contact` controller's `store` method). We're going to pass in an array containing the contact form's name, e-mail address, and message values, so let's modify the constructor to assign the array to a class instance variable:

[76] https://github.com/wjgilmore/hackerpair/blob/master/app/Mail/ContactEmail.php

```
public $contact;

public function __construct($contact)
{
    $this->contact = $contact;
}
```

The `build` method is responsible for generating and delivering the e-mail, identifying the recipient address, e-mail subject, and view used for the e-mail content (among other things). The HackerPair `build` method looks like this:

```
public function build()
{
    return $this
        ->to(config('mail.from.address'))
        ->subject('HackerPair Inquiry')
        ->view('emails.contact');
}
```

The e-mail template is found in `resources/views/emails/contact.blade.php`. It is very simple, containing just a brief introductory note and the form values:

```
Hi,

A HackerPair user has sent you a message.

Name: {{ $contact['name'] }}

E-mail: {{ $contact['email'] }}

Message: {{ $contact['msg'] }}
```

With the template in place, there's just one more step. Invoke the `ContactEmail` class! Open the `Contact` controller and replace the mail-related placeholder with this:

```
Mail::to(config('mail.support.address'))->send(new ContactEmail($contact));
```

Don't forget to reference the class at the top of the file:

```
use App\Mail\ContactEmail;
```

After saving the changes, fill out and submit the form. If everything is properly in place, you should receive an e-mail at the recipient address within a few moments.

If you neglected my advice against using Gmail and are, experiencing problems, it could be because of a Gmail setting pertaining to third-party access to your account. Enable the "Less secure apps" setting at https://www.google.com/settings/security/lesssecureapps[77] to resolve the issue. Even if this resolves the issue, keep in mind you nonetheless should not use your Gmail account for production purposes.

Protecting Your Form with a CAPTCHA

TODO: Readers, this section is not yet complete. I'll update it soon!

These days, any sort of publicly available web form is guaranteed to be a target of spammers and other bots. Several years ago a very effective solution for thwarting automated spammers known as a CAPTCHA was devised. This requires anybody (human or bot) to first complete a challenge before the form can be submitted. You've undoubtedly encountered more than a few CAPTCHA challenges when completing web forms.

```
$ composer require anhskohbo/no-captcha
```

The no-captcha package supports Laravel 5.5's auto-discovery feature, so you only need to modify the `config/app.php` file if you're running Laravel 5.4 or older. If you fall into the latter circumstance, add the following line to the `config/app.php` `providers` array:

```
Anhskohbo\NoCaptcha\NoCaptchaServiceProvider::class,
```

Finally, add the following line to the `config/app.php` `aliases` array:

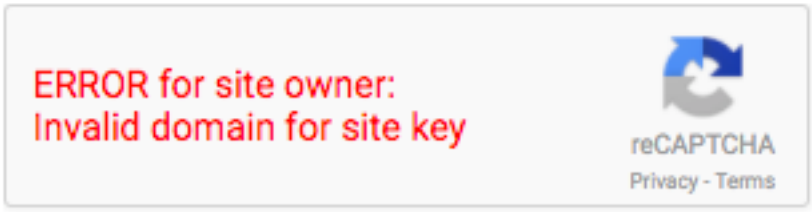
```
'NoCaptcha' => Anhskohbo\NoCaptcha\Facades\NoCaptcha::class,
```

Regardless of what version of Laravel you're running, you'll need to generate a reCAPTCHA API site key and secret key. Go to https://www.google.com/recaptcha/admin to generate these keys. On this page you'll also have to choose the type of reCAPTCHA you'd like to incorporate into the forms:

- reCAPTCHA V2:

[77] https://www.google.com/settings/security/lesssecureapps

- Invisible reCAPTCHA:
- reCAPTCHA Android:



Invalid domain error

Changing the Default Captcha Settings

You can optionally change various default Captcha settings (length, size, etc.) by publishing the package's configuration file:

```
1 $ php artisan vendor/publish
```

This will create a new file named `captcha.php` in the `config` directory.

Summary

This was one of the more entertaining chapters to write because it really brings together multiple Laravel capabilities to create an immediately useful feature. In the next chapter, you'll learn how to incorporate user accounts into your application, really taking the opportunities for interaction to the next level!

Chapter 9. Authenticating and Managing Your Users

Integrating user accounts into your application opens up a whole new world of possibilities in terms of enhanced interaction through customized features. While in some cases account registration is optional, it's typically a requirement for users who desire to interact with the application in a meaningful way beyond page navigation. Such is the case for HackerPair users, because only registered users can create and join events, not to mention perform auxiliary tasks such as favorite events.

Because good things rarely come easy, there's quite a bit to keep in mind when incorporating user accounts into a web application, including: the user registration process, secure storage of user credentials, authentication, logout, authorization (some users might possess greater privilege levels than others), password recovery, and profile management.

Fortunately, Laravel removes numerous headaches associated with implementing many of these features! In this chapter you'll learn how to integrate Laravel's native authentication features into your own application. You'll also learn how to integrate third-party authentication providers such as Facebook, GitHub, and Twitter into your application, allowing users to join your application by authenticating their identity using these trusted and well-known services.

Registering Users

Implementing the user registration feature seems to be a logical starting point. Although it's fairly straightforward, this section is easily the longest in the chapter because there are a few other matters I necessarily need to introduce, beginning with the model used to manage the user accounts.

Introducing the User Model and Users Table

You're spared the hassle of building a model and underlying table for managing user accounts, because all new Laravel applications include a `User` model for this purpose. You'll find it in `app/User.php`, and it looks like this:

```php
<?php

namespace App;

use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use Notifiable;

    protected $fillable = [
        'name', 'email', 'password',
    ];

    protected $hidden = [
        'password', 'remember_token',
    ];
}
```

This `User` model looks rather different from the models we've created in earlier chapters, notably because it extends the `User` (you'll see at the top of the file `Illuminate\Foundation\Auth\User` is aliased as `Authenticatable`) class rather than `Model`. This `User` class in turn implements several *contracts* and uses several *traits*. A contract defines an interface to a particular implementation of a set of features. For instance, if you dig into the Laravel framework source code and open the `Illuminate\Foundation\Auth\User` class[78], you'll see it implements the `AuthenticatableContract`, `AuthorizableContract`, and `CanResetPasswordContract`. The `AuthenticatableContract` defines an interface for obtaining the user's unique identifier and password and for managing the "remember me" token should it be enabled. The `AuthorizableContract` defines an interface for determining whether the user is allowed to perform a certain task as defined by an "ability". Laravel's Authorization feature isn't covered in the book presently, but I'll be sure to add a chapter on the topic in the near future. Finally, the `CanResetPasswordContract` defines an interface for resetting a user's lost password.

The contracts work in unison with the *traits*. As you can see, the `Illuminate\Foundation\Auth\User` model uses three traits, including `Authenticatable`, `Authorizable`, and `CanResetPassword`. Traits offer a useful alternative to multiple inheritance (something PHP can't do natively), allowing you to inherit methods from several different classes, thereby avoiding code duplication. Therefore the contract defines the interface, and the trait identifies the interface implementation. This means you could for instance replace the implemented traits with your own, provided you meet the requirements defined in the contract.

[78] https://github.com/laravel/framework/blob/master/src/Illuminate/Foundation/Auth/User.php

Philip Brown penned a great introductory tutorial[79] to traits. It's worth taking the time to read now if this concept is new to you.

I've gone into all of this additional detail to make a point clear: Laravel has already taken care of *a lot* of the gory details associated with user account management. This allows us to assemble these features in a manner more akin to putting together puzzle pieces than writing a great deal of code.

Returning to the `app/User.php` model, you'll see an additional `Notifiable` trait is used. This feature allows you to send brief messages to users pertaining to application-related updates, such as when a HackerPair user joins or leaves an event, or when a new event is created in a user's zip code. Like authorization, I'll be updating the book soon to include coverage of notifications.

Next you'll see the `$fillable` property has been defined. It identifies the columns that can be inserted/updated by way of mass assignment, and is currently set to allow mass assignment of a user's `name`, `email`, and `password` fields. Finally, the `$hidden` property is used to identify columns that should not be passed into JSON or arrays. Logically we don't want to expose the password (even if in hashed format) nor the session remember token, and so these are identified in the `$hidden` property.

Introducing the Users Table

In addition to the `User` model, you'll also find a corresponding `users` table migration, located in `database/migrations/2014_10_12_000000_create_users_table.php`. Because by this point in the book you likely already ran `php artisan migrate`, the `users` table already exists in your project database. The table looks like this:

```
mysql> describe users;
+----------------+------------------+------+-----+...
| Field          | Type             | Null | Key |...
+----------------+------------------+------+-----+...
| id             | int(10) unsigned | NO   | PRI |...
| name           | varchar(255)     | NO   |     |...
| email          | varchar(255)     | NO   | UNI |...
| password       | varchar(60)      | NO   |     |...
| remember_token | varchar(100)     | YES  |     |...
| created_at     | timestamp        | NO   |     |...
| updated_at     | timestamp        | NO   |     |...
+----------------+------------------+------+-----+...
7 rows in set (0.01 sec)
```

With the `User` model and `users` table in place, let's figure out how to create a registration form and wire it up to some registration logic. Fortunately this is surprisingly easy thanks to some additional Laravel automation.

[79] http://culttt.com/2014/06/25/php-traits/

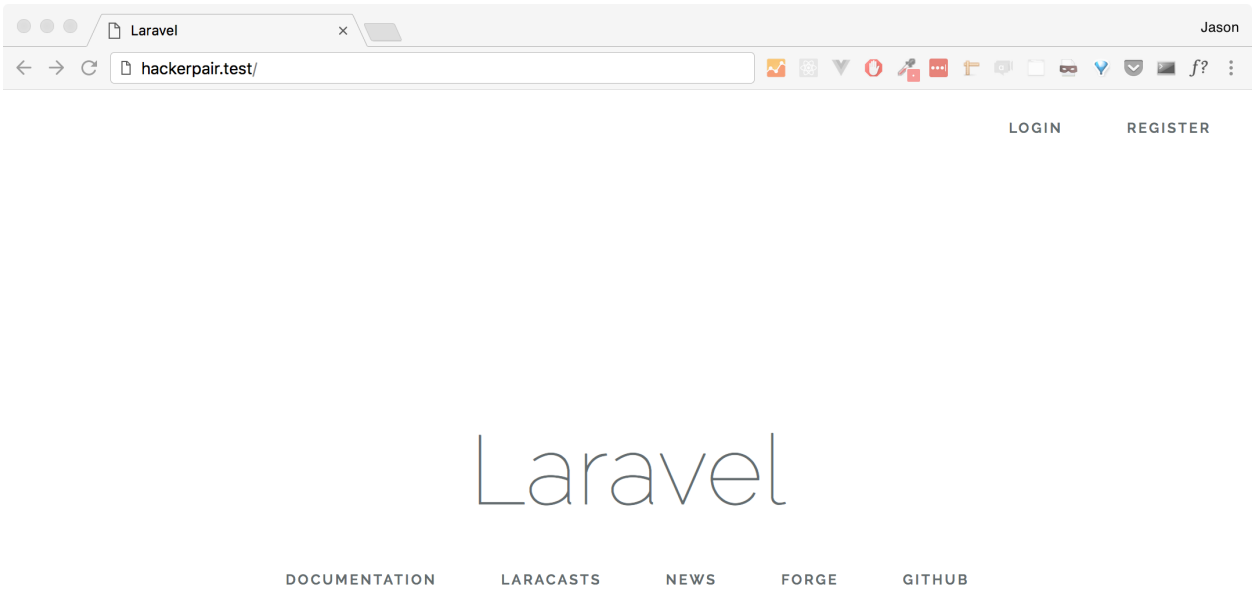
Generating the Authentication Scaffolding

Despite being perfectly capable of carrying out the various account registration, sign in, sign out, and password recovery tasks, your Laravel application doesn't by default include the associated forms and routes. This is because the Laravel team doesn't want to unnecessarily clutter an application skeleton with features that might wind up going unused. However, you can easily generate the routes, views, and forms by executing Artisan's `make:auth` command. Before doing so however, understand that if you've already started work on a home page and happen to have chosen the name `home.blade.php`, or have created a layout called `app.blade.php` inside the `views/layouts` directory, running this command will *completely* replace any changes you've made to these files! Therefore take care to create a copy of the files before running the command (or preferably just commit your changes to version control and then revert the changes):

```
1 $ php artisan make:auth
2 Authentication scaffolding generated successfully.
```

In addition to creating a slate of new views inside the directory `resources/views/auth`, you'll see the aforementioned `welcome.blade.php` view and `layouts/app.blade.php` layout have been generated. Additionally, you'll find a confusingly named view called `home.blade.php`. We'll return to this latter file in just a bit, so don't worry about it for now.

Presuming `welcome.blade.php` is still responsible for your project's home page, and it's using the `app.blade.php` layout, return to the browser and reload the site. You'll see a rather different home page than what was presented in chapter 1:



The refactored home page

In addition, open `routes/web.php` and you'll find the following two new route definitions:

```
1  Auth::routes();
2
3  Route::get('/home', 'HomeController@index')->name('home');
```

Just as `Route::resource()` is a shortcut for defining the seven RESTful routes, so is `Route::auth()` for defining the following routes:

```
Route::get('login', 'Auth\LoginController@showLoginForm')->name('login');
Route::post('login', 'Auth\LoginController@login');
Route::post('logout', 'Auth\LoginController@logout')->name('logout');

// Registration Routes...
Route::get('register', 'Auth\RegisterController@showRegistrationForm')
    ->name('register');
Route::post('register', 'Auth\RegisterController@register');

// Password Reset Routes...
Route::get('password/reset', 'Auth\ForgotPasswordController@showLinkRequestForm')
    ->name('password.request');
Route::post('password/email', 'Auth\ForgotPasswordController@sendResetLinkEmail')
    ->name('password.email');
Route::get('password/reset/{token}', 'Auth\ResetPasswordController@showResetForm\
')
    ->name('password.reset');
Route::post('password/reset', 'Auth\ResetPasswordController@reset');
```

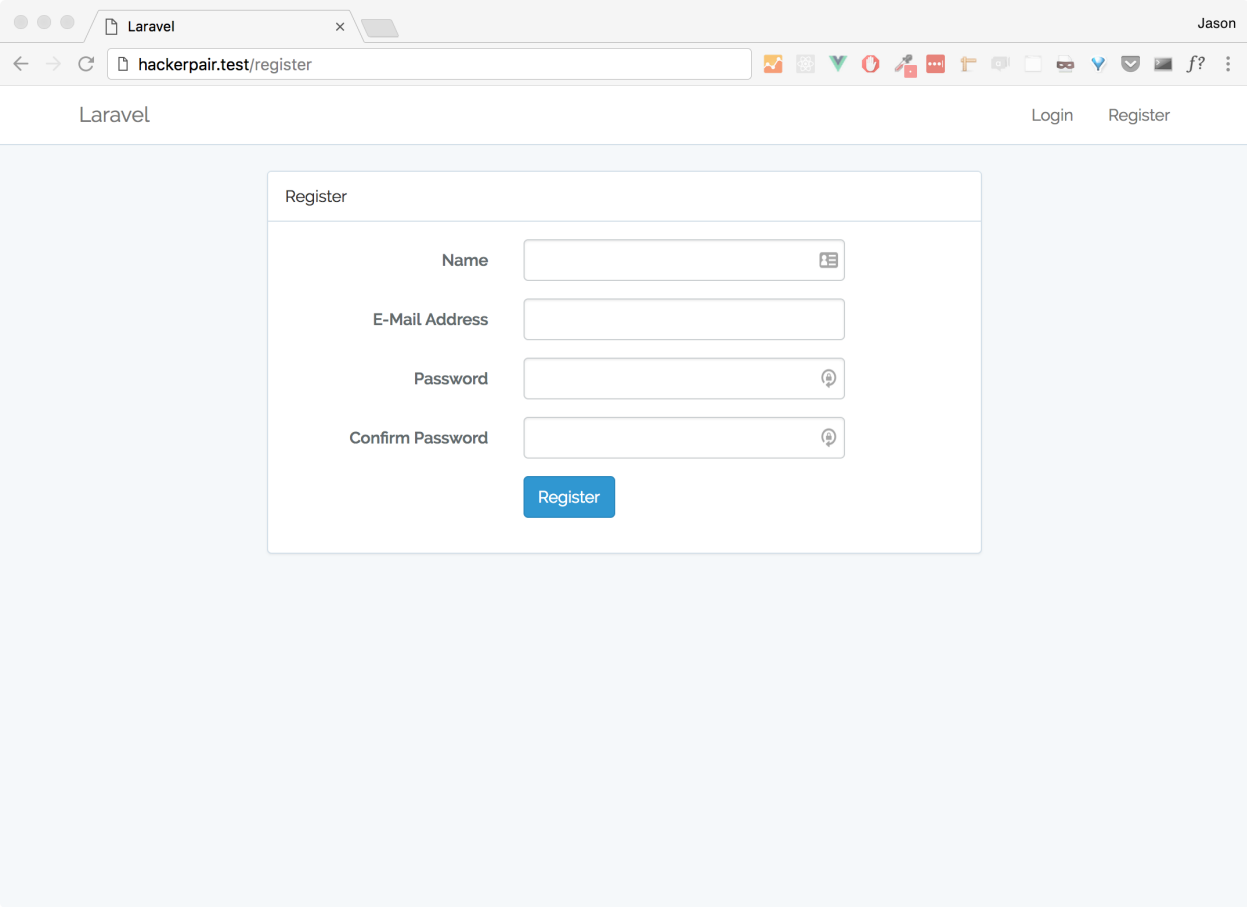
You'll become well-acquainted with these routes in the sections to follow.

Integrating Account Registration

After running `make:auth`, your application's `routes/web.php` file will include a reference to `Auth::routes()`, which makes available the following two registration-related routes:

```
Route::get('register', 'Auth\RegisterController@showRegistrationForm')
    ->name('register');
Route::post('register', 'Auth\RegisterController@register');
```

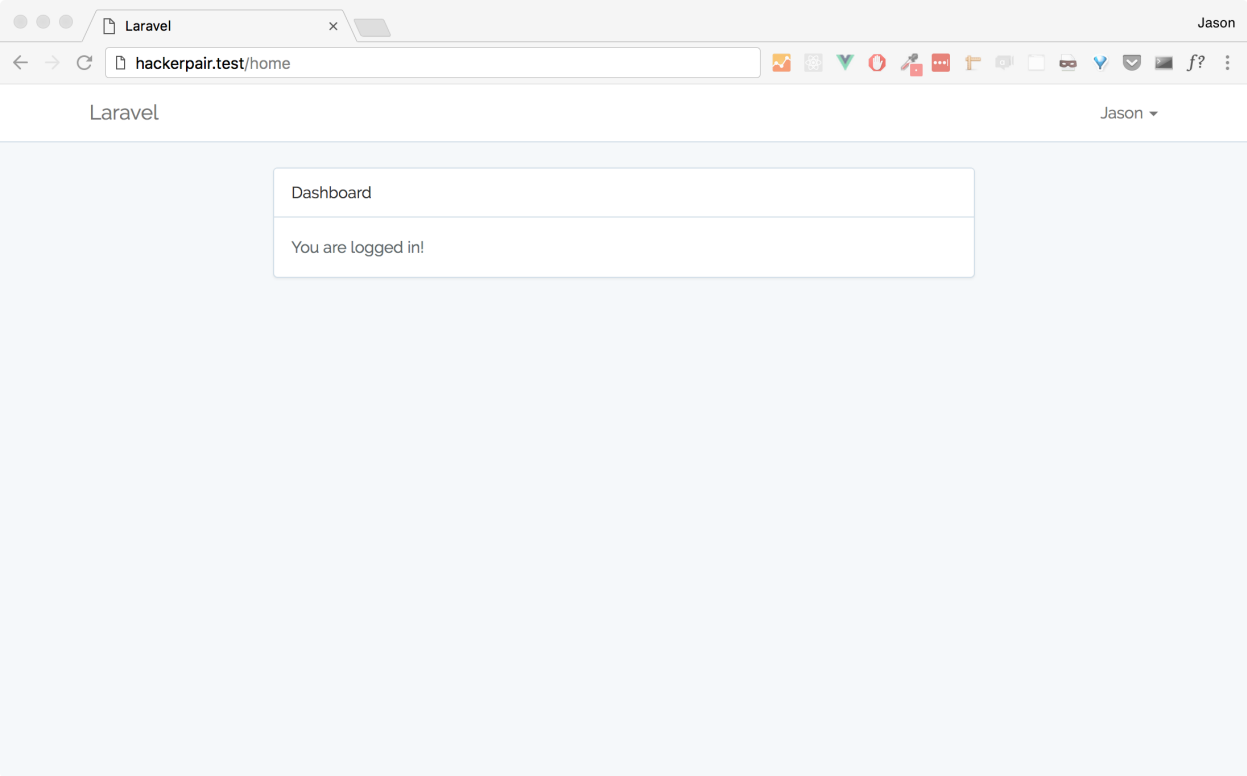
The `get`-based route is responsible for presenting the registration form, while the `post`-based route is responsible for processing the form input. These routes point to `showRegistrationForm()` and `register()` methods found in the `app/Http/Controllers/Auth/RegisterController.php` file, respectively. However, if you open this controller you won't actually find these methods, because they are made available by way of the `RegistersUsers` trait, which in turn makes them available by way of the `RegistersUsers` trait located in `vendor/laravel/framework/src/Illuminate/Foundation/Auth/RegistersUsers.php`. I realize this probably sounds a tad confusing, but I suggest you have a look at the `RegistersUsers.php` file (where you will find the `showRegistrationForm()` and `register()` methods), and everything will start to make sense. Regardless of whether you do, rest assured your application can now successfully register users. In fact if you return to the home page and click the `Register` link, you'll be presented with the following form:



The default registration page

Presuming you configured the project database as described in chapter 3, you can successfully register a test user simply by completing this form. That's right, there is *no additional coding* required to begin accepting user registrations! After creating an account, have a look at the project database's `users` table and you'll see the new record.

Further, Laravel will automatically sign you into the site and attempt to redirect to the `/home` URI. As you can see from the following screenshot, newly registered users are additionally signed in (note my name is displayed at the top right of the page):



Newly registered users are automatically signed in

This destination is managed by the following route, which was added to the `web.php` file when you executed `make:auth`:

```
1 Route::get('/home', 'HomeController@index')->name('home');
```

Chances are you'll want to change this post-authentication destination. To do so, you can either point the above `/home` route to a different controller and action (and corresponding view), or you can override the redirection URI by modifying the `$redirectTo` property within the `RegisterController.php` file:

```
1 protected $redirectTo = '/';
```

Integrating Account Sign In

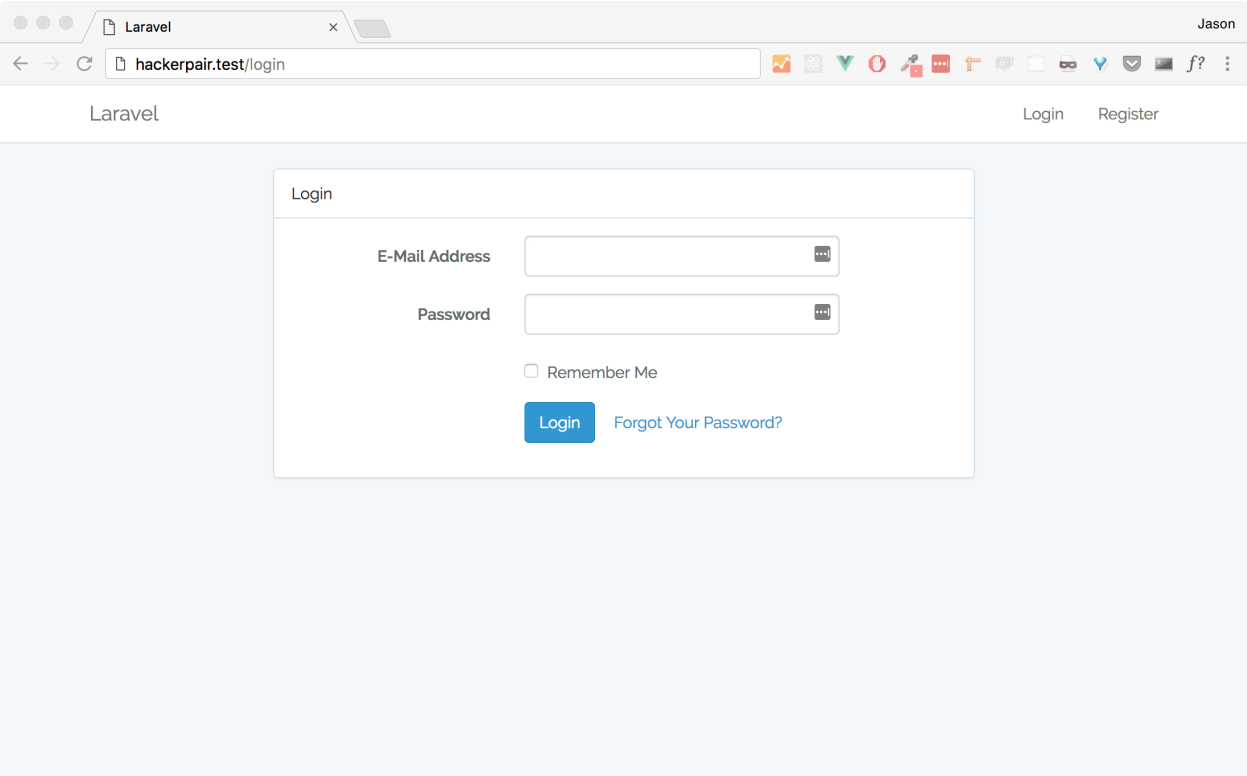
Newly registered users are automatically signed in following registration, however logically those users will want to end their session and sign out. Hopefully they'll return, and so you'll want to provide an easy solution for signing into their account.

When `make:auth` was executed, the following two authentication-related routes will be made available via the `Route::auth()` shortcut:

```
Route::get('login', 'Auth\LoginController@showLoginForm')->name('login');
Route::post('login', 'Auth\LoginController@login');
```

Like the registration routes, the `get`-based route is responsible for displaying the login form, while the `post`-based route is responsible for processing the form input and determining whether the credentials are valid. Also, while the `app/Http/Controllers/Auth/LoginController.php`'s `showLoginForm` and `login` methods are identified as being responsible for these respective endpoints, you won't actually find the methods in `AuthController.php` because they are instead made available via the `AuthenticatesUsers.php` trait (located in `vendor/laravel/framework/src/Illuminate/Foundation/Auth`).

You can give authentication a whirl by first logging out of your newly registered account (click the `Logout` link located in the navigation bar), which will end your session and return you to the `welcome.blade.php` view. Next, click on `Login` (also in the navigation bar) where you'll be greeted with the form presented in the following screenshot:



The default sign in view

Sign in using the account you created in the last section, and as before you'll be redirected to the `Home` controller. Like the registration form, you can easily modify the sign in view to suit your needs by editing the view found at `resources/views/auth/login.blade.php`.

Just as you can override the default destination route for newly registered users, so can you do so for newly authenticated users. You can either override the default controller used for the `/home` URI in

`web.php`, or override the default post-authentication redirection URI by modifying the `$redirectTo` property in the `Login` controller (found in `app/Http/Controllers/Auth/LoginController.php`):

```
protected $redirectTo = '/';
```

Enabling Authentication Throttling

A Laravel feature called *authentication throttling* which will prevent further authentication attempts for a predefined period of time if the user fails multiple sign in attempts. Throttling is disabled by default, however if you'd like to enable it, open your `LoginController` class (`app/Http/Controllers/Auth/LoginController.php`) and add the `ThrottlesLogins` trait:

```
use Illuminate\Foundation\Auth\ThrottlesLogins;
...

class LoginController extends Controller {

  use AuthenticatesAndRegistersUsers, ThrottlesLogins;
  ...

}
```

If you want to change the default number of login attempts, add the `maxAttempts` property to your `Login` controller:

```
protected $maxAttempts = 3;
```

If you want to change the number of minutes a user must wait after reaching the maximum number of allowable login attempts, add the `decayMinutes` property to your `Login` controller:

```
protected $decayMinutes = 5;
```

Integrating Account Sign Out

All Laravel applications include the ability to end an authenticated session by signing out of an account. The `Auth::routes()` shortcut adds the following route to your application:

```
Route::post('logout', 'Auth\LoginController@logout')->name('logout');
```

Take careful note this is a POST-based route, meaning you can't just create a hyperlink to `/logout` and expect it to work. For instance, if you open the `resources/views/layouts/app.blade.php` file you'll see the `logout` link is actually implemented like this:

```
<a href="{{ route('logout') }}"
    onclick="event.preventDefault();
             document.getElementById('logout-form').submit();">
    Logout
</a>

<form id="logout-form" action="{{ route('logout') }}" method="POST" style="displ\
ay: none;">
    {{ csrf_field() }}
</form>
```

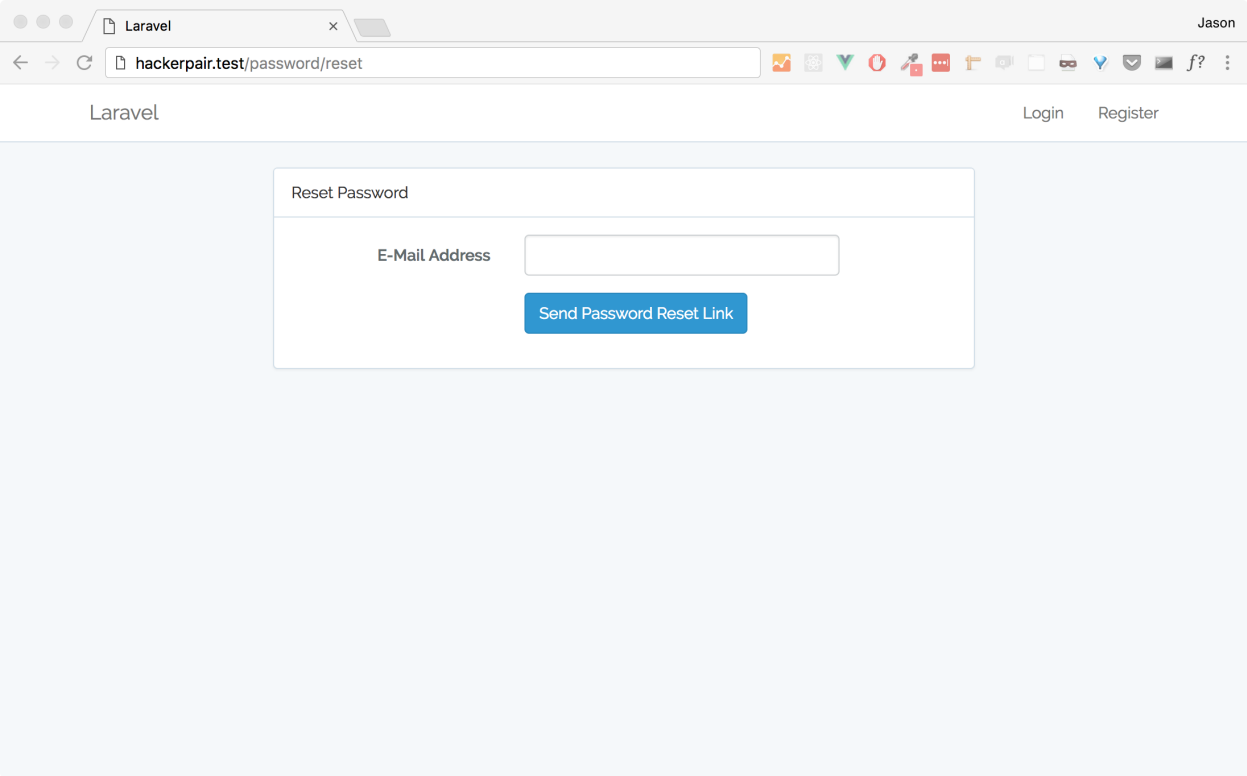
If you return to the browser, you'll see the `Logout` link is rendered as a hyperlink. The above snippet ensures that when the link is clicked, the hidden form identified by the ID `logout-form` is submitted. It's this bit of trickery which allows the POST-based logout route to be executed.

Password Recovery

A convenient password recovery is also available after generating the authentication routes and views. The following three routes are made available via the `Route::auth()` shortcut:

```
Route::get('password/reset', 'Auth\ForgotPasswordController@showLinkRequestForm')
    ->name('password.request');
Route::post('password/email', 'Auth\ForgotPasswordController@sendResetLinkEmail')
    ->name('password.email');
Route::get('password/reset/{token}', 'Auth\ResetPasswordController@showResetForm\
')
    ->name('password.reset');
Route::post('password/reset', 'Auth\ResetPasswordController@reset');
```

You can see the default recovery form by signing out of your account (via `/logout`) and after being returned to the home page, click `Login` and then the `Forgot Your Password?` link. You'll be taken to the password recovery form presented in the following screenshot:



Initiating password recovery

If you've configured your application's mail driver (see chapter 8), and the e-mail address exists in the `users` table, then a password recovery e-mail will be sent to the address you specified. Review this e-mail and you'll see a one-time URL is sent to the user. When the user clicks the link found in the e-mail, Laravel will consult the project database's `password_resets` table (like `users`, this table will automatically be created when you run migrations for the first time). Here's an example record found in the `password_resets` table:

```
mysql> select * from password_resets;
+------------------+--------------------+---------------------+
| email            | token              | created_at          |
+------------------+--------------------+---------------------+
| wj@wjgilmore.com | asdfasfaasfasfasdf | 2018-01-04 19:26:02 |
+------------------+--------------------+---------------------+
1 row in set (0.00 sec)
```

When the user clicks this link, Laravel will consult `password_resets` for a matching e-mail address and token, and if they match the user will be able to choose a new password. As you can see, the default recovery e-mail is quite sparse, however you can update it to include whatever additional information you please provided the recovery link formatting remains intact.

This e-mail won't be successfully sent until you configure `config/mail.php`. See chapter 8 for more information about configuring Laravel's e-mail delivery feature. Alternatively, set the `.env` file's `MAIL_DRIVER` setting to `log` and tail the `storage/logs/laravel.log` file to view the mail output.

The recovering user will have 60 minutes to click on the recovery link per the `config/auth.php` file's `passwords['users']['expire']` setting. You can change this setting to whatever value you desire. For instance to give users up to 24 hours to recover the password, you'll set `expire` to `1440`.

Retrieving the Authenticated User

You can retrieve the `users` record associated with the authenticated user via `Auth::user()`. For instance, to retrieve the authenticated user's name you'll access `Auth::user()` like so:

```
1  Welcome back, {{ Auth::user()->name }}!
```

However, you'll want to ensure the user is authenticated before attempting to access the user object. You can do so via one of several approaches. One approach involves using the Blade's convenient `@auth` directive:

```
1  @auth
2    Welcome back, {{ Auth::user()->name }}!
3  @else
4    Hello, stranger! <a href="{{ route('login') }}">Login</a>
5    or <a href="{{ route('register') }}">Register</a>.
6  @endauth
```

Alternatively, you can use the `Auth` facade's `check` method inside a conditional:

```
1  @if (Auth::check())
2    Welcome back, {{ Auth::user()->name }}!
3  @else
4    Hello, stranger! <a href="{{ route('login') }}">Login</a>
5    or <a href="{{ route('register') }}">Register</a>.
6  @endif
```

Conversely, you can flip the conditional around, instead using the `Auth` facade's `guest` method to determine if the user is a guest:

```
@if (Auth::guest())
  Hello, stranger! <a href="{{ route('login') }}">Login</a>
  or <a href="{{ route('register') }}">Register</a>.
@else
  Welcome back, {{ Auth::user()->name }}!
@endif
```

Restricting Forms to Authenticated Users

Recall from chapter 8 that all form request classes include a method named `authorize`. This method is used to determine the circumstances in which the form can be submitted. By default it is set to `false`, and so in chapter 8 we updated it to instead return `true` because we had not yet integrated user accounts. To refresh your memory here's what a default `authorize` method looks like inside a newly generated form request:

```
<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class ContactFormRequest extends FormRequest {

    public function authorize()
    {
        return false;
    }

    ...

}
```

If you'd like to restrict a form request to authenticated users, you can modify the `authorize` method to look like this:

```
public function authorize()
{
    return Auth::check();
}
```

Keep in mind you're free to embed into `authorize` whatever logic you deem necessary to check a user's credentials. For instance if you wanted to restrict access to not only authenticated users but additionally only those who are paying customers, you can retrieve the user using the `Auth` facade's `user` method and then traverse whatever associations are in place to determine the user's customer status.

Adding Custom Fields to the Registration Form

The default registration form only includes fields for the user's name, e-mail address, and password. However, what if you wanted to additionally ask for a username or perhaps a location such as the user's country? Fortunately, Laravel's authentication implementation makes this an incredibly easy task. I'll walk you through an example in which we require the user to additionally provide a unique username. Begin by updating the `users` table to include a `username` field:

```
$ php artisan make:migration add_username_to_users_table --table=users
```

Open the newly created migration file and modify the `up` method to look like this:

```
Schema::table('users', function(Blueprint $table)
{
    $table->string('username');
});
```

Modify the `down` method to look like this:

```
Schema::table('users', function(Blueprint $table)
{
    $table->dropColumn('username');
});
```

Run the migration and then open the `resources/auth/register.blade.php` form we created earlier in the chapter. Add a field for accepting the username:

```
<div class="form-group">
    <label class="col-md-4 control-label">Username</label>
    <div class="col-md-6">
        <input type="text" class="form-control"
          name="username" value="{{ old('username') }}">
    </div>
</div>
```

Of course, Laravel won't automatically know what to do with this field, therefore you'll need to make one last change. Open the `AuthController.php` controller found in `app/Http/Controllers/Auth`, and modify the `validator` method to look like this:

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

I've emphasized the line you'll need to add. Next, scroll down and modify the `create` method to look like this:

```
protected function create(array $data)
{

    return User::create([
        'name' => $data['name'],
        'email' => $data['email'],
        'username' => $data['username'],
        'password' => bcrypt($data['password']),
    ]);
}
```

Save these changes, and only one step remains. Open the `app/User.php` file and add the `username` field to the `$fillable` property like so:

```
protected $fillable = [
    'name', 'email', 'password', 'username'
];
```

Save these changes, and believe it or not you're ready to register users with usernames! It really is that easy. Read the next section if you want to require users to login using their username rather than their e-mail address.

Allowing Login Using a Username

Laravel's default authentication scheme is configured to require users to supply an e-mail address and password. It's however often the case that you'll want to instead ask for a username and password. Presuming you've added the `username` field to your database and required the user to supply a username during registration, as described in the previous section, changing this requirement is incredibly easy. Begin by opening your `resources/views/auth/login.blade.php` view and modifying the e-mail block to instead ask for a username:

```
<div class="form-group{{ $errors->has('username') ? ' has-error' : '' }}">
    <label for="username" class="col-md-4 control-label">Username</label>

    <div class="col-md-6">
        <input id="username" type="username" class="form-control"
          name="username" value="{{ old('username') }}" required autofocus>

        @if ($errors->has('username'))
            <span class="help-block">
                <strong>{{ $errors->first('username') }}</strong>
            </span>
        @endif
    </div>
</div>
```

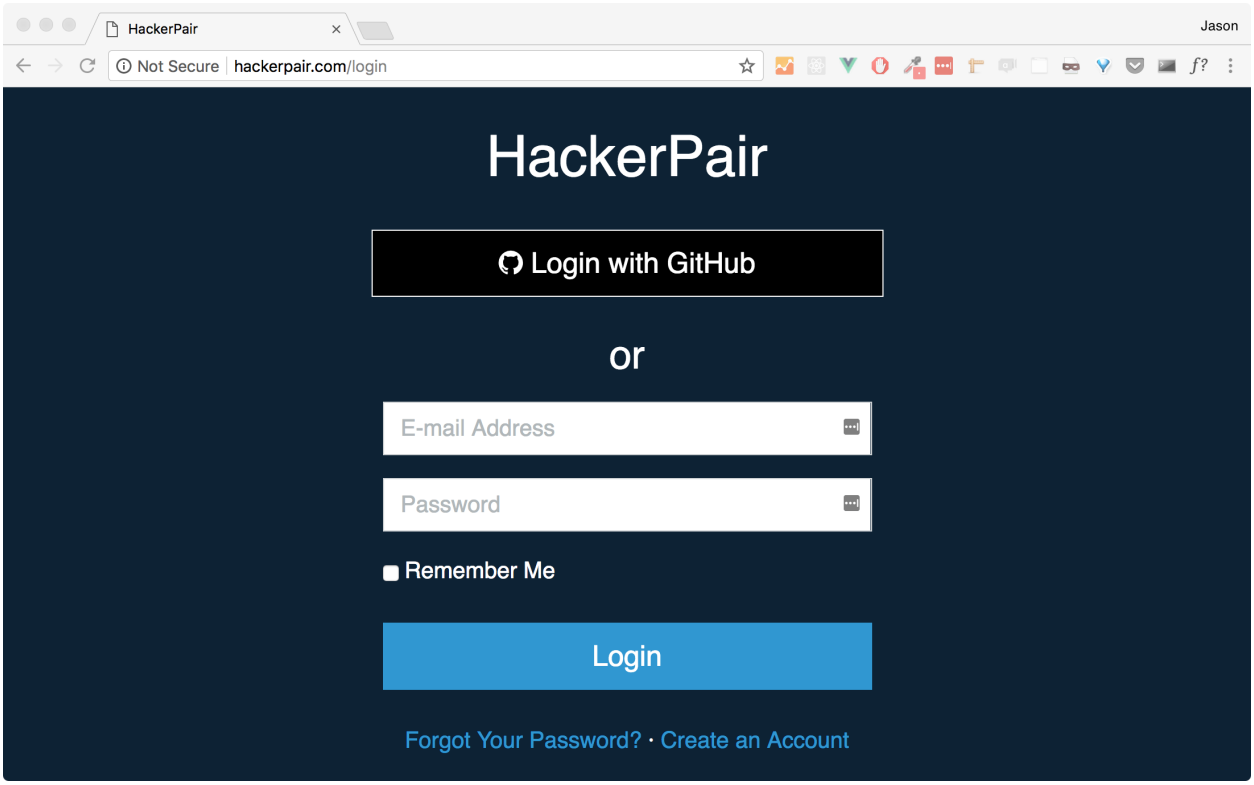
Save the changes and open your `app/Http/Controllers/Auth/LoginController.php` file and add the following method to the `LoginController` class:

```
public function username()
{
    return 'username';
}
```

Save your changes and return to the login form. Provided you've populated the user record with a username, you'll be able to login using your account username and password!

Integrating OAuth with Laravel Socialite

Head on over to the HackerPair website (http://hackerpair.com) and click the `Login` link (see below screenshot). You'll see users can login either with a standard account or using their GitHub credentials.

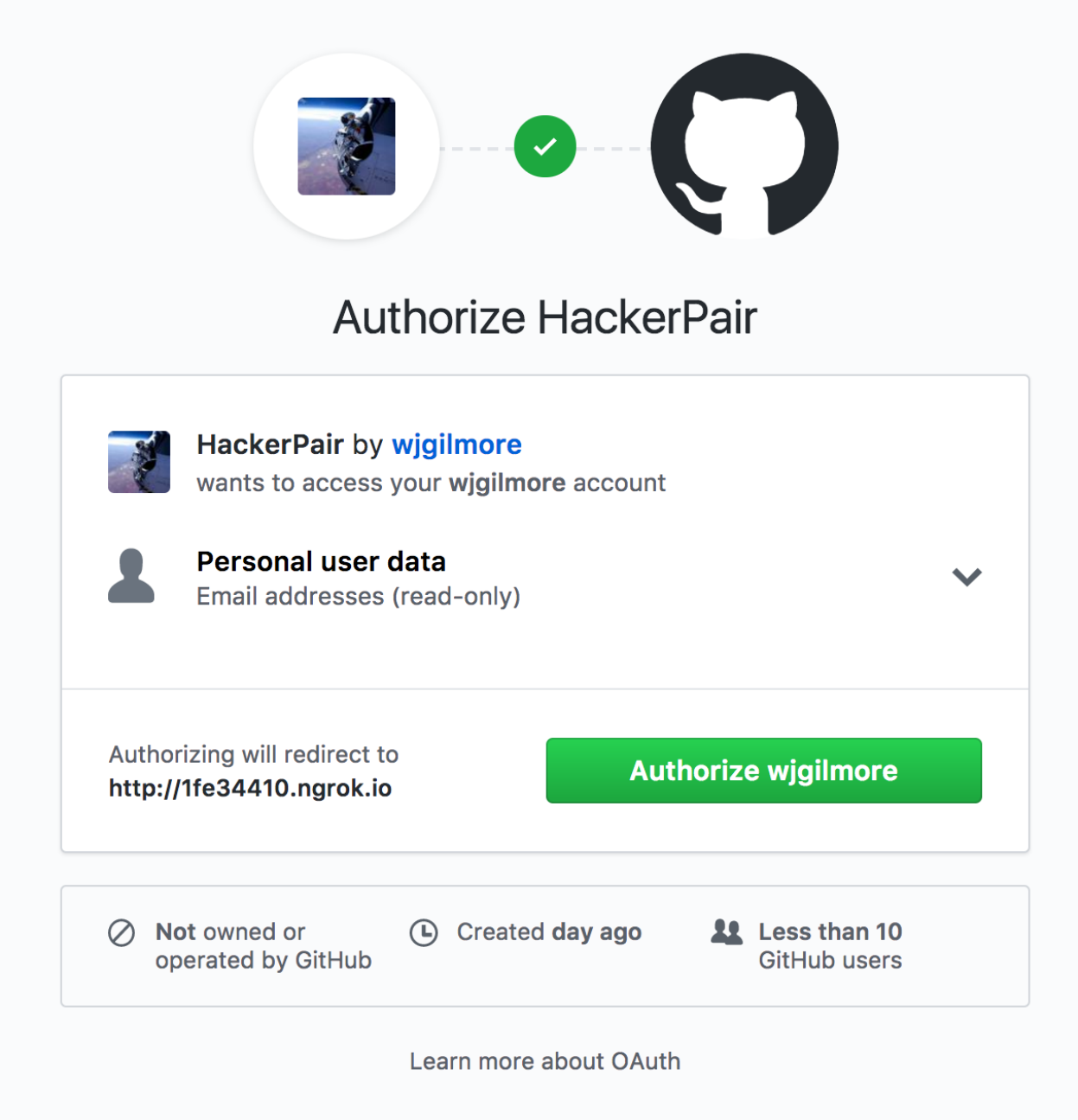


Logging into HackerPair with a standard account or GitHub

Many well-known online service providers such as GitHub, Twitter, and Amazon offer third-parties the opportunity to provide users with the ability to register and authenticate their identity using accounts managed by these service providers rather than creating and managing yet another account. As an added bonus, users can choose this option without ever having to expose their account credentials. This is possible thanks to an open standard known as OAuth[80].

Presuming you have a GitHub account, I invite you to try logging in by clicking on the GitHub button. When you do you'll be greeted with the authorization screen presented in the below screenshot:

[80] https://en.wikipedia.org/wiki/OAuth



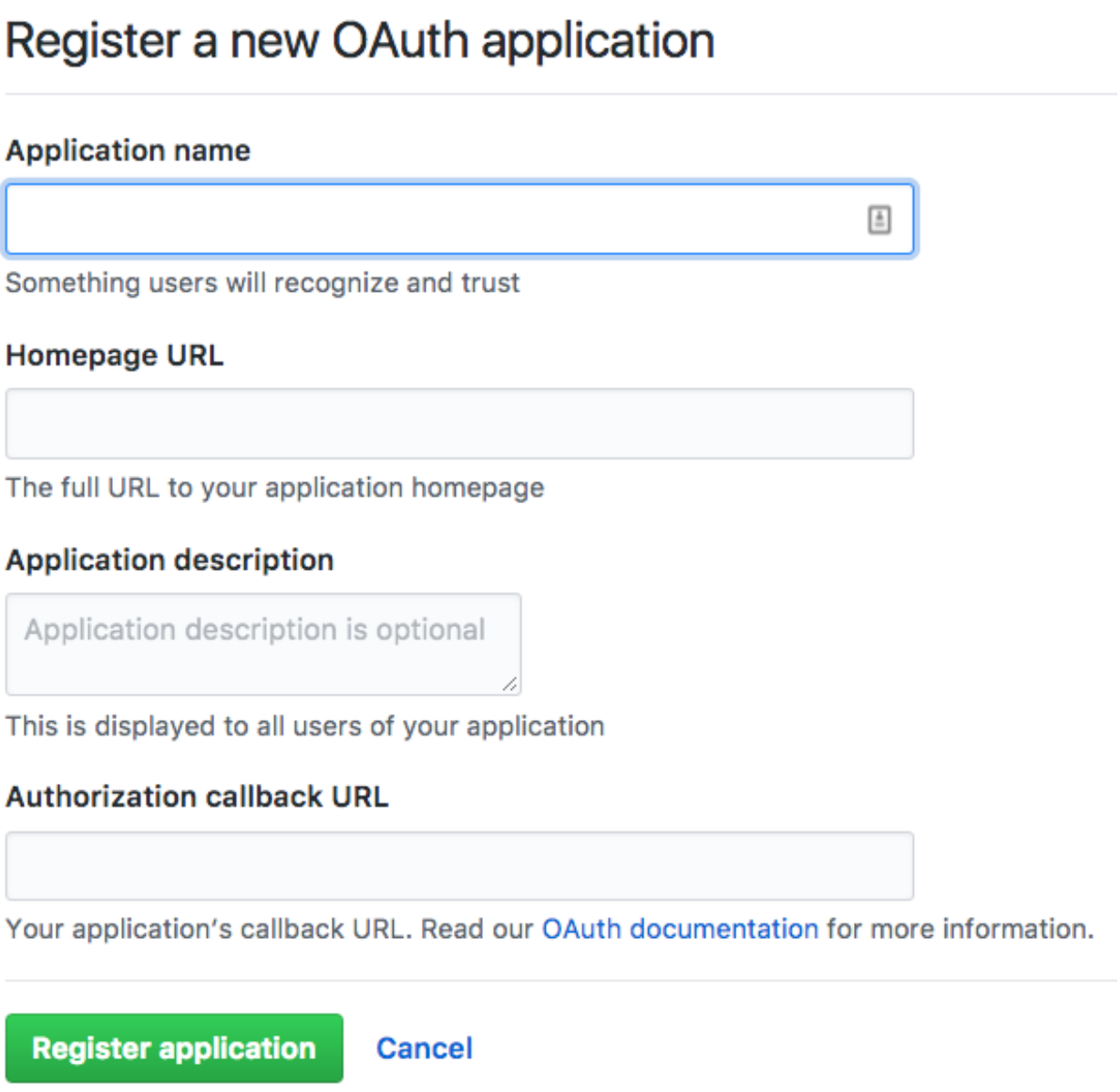
The GitHub authentication authorization screen

Just click the `Authorize wjgilmore` button (that's me) and you'll be returned to the HackerPair home page and presented with the message `Successfully authenticated using GitHub`.

Laravel offers a package called Socialite which takes care of the many otherwise gory details you'd have to handle when adding OAuth-based authentication you a web application. Begin by adding the Socialite package to your application:

```
$ composer require laravel/socialite
```

Socialite supports a number of different OAuth providers, among them Facebook, GitHub, Google, and Twitter, however I'm going to focus on GitHub. The beauty of Socialite though is it is trivial to adapt what you learn here to any of the other providers. To configure GitHub OAuth authentication, you'll first need to login to your GitHub account, click on `Settings`, and then click on `Developer Settings`. The `OAuth Apps` menu items should be selected by default. Click the `New OAuth App` button and you'll be presented with the form found in the following screenshot.



Registering a new OAuth application

Frankly, the first three fields aren't useful for anything more than informational purposes. The `Application callback URL` field is however crucial. This URL is used by the OAuth provider as the return destination following authentication. Therefore presuming you're developing the project locally using an environment such as Homestead or Valet, you can't just enter a value such as `http://hackerpair.test` and expect it to work. Instead, you'll need to expose your development site to the public web. If you're using Valet, the easiest way to do so is by way of Valet's sharing feature. Otherwise, you'll need to use a service such as ngrok[81] (which is coincidentally the same service Valet uses) or another solution capable of exposing your local site to the public network.

After registering the application, you'll be provided with a client ID and secret. Open your `.env` file and assign these values and your designated callback like so:

```
GITHUB_CLIENT_KEY=CLIENT_ID_GOES_HERE
GITHUB_CLIENT_SECRET=CLIENT_SECRET_GOES_HERE
GITHUB_CALLBACK=CALLBACK_URL_GOES_HERE
```

Save this change and then open `config/services.php`, adding the following array to it:

```
'github' => [
    'client_id'     => env('GITHUB_CLIENT_KEY'),
    'client_secret' => env('GITHUB_CLIENT_SECRET'),
    'redirect'      => env('GITHUB_CALLBACK'),
],
```

This configuration data will be used by the local authentication logic to initiate the OAuth-based GitHub login process.

Integrating the OAuth Logic

With the configuration-related tasks completed, it's time to integrate the authentication logic into your application. Perhaps in a future revision I'll explain how to properly abstract this code to support multiple providers, but for the moment let's just focus on GitHub. Begin by creating a new migration which will add a few new fields to the `users` table:

```
$ php artisan make:migration add_oauth_fields_to_users --table=users
```

Open the newly created migration file and modify the `up` method to look like this:

[81] https://ngrok.com/

```
Schema::table('users', function (Blueprint $table) {
    $table->integer('provider_id')->unsigned()->nullable();
    $table->string('handle_github')->nullable();
});
```

Modify the `down` method to look like this:

```
public function down()
{
    Schema::table('users', function (Blueprint $table) {
        $table->dropColumn('provider_id');
        $table->dropColumn('handle_github');
    });
}
```

The `provider_id` field acts as a foreign key for the OAuth provider, used to match any previously authenticated users with their local database records. The `handle_github` field will contain the user's GitHub username. This information is exposed by the GitHub OAuth API, and so upon initial registration we'll retrieve this value and store it in the database for later use (such as pointing fellow HackerPair users to the registered user's GitHub page).

After running the migration, create a new controller which will manage the OAuth-specific logic:

```
$ php artisan make:controller SocialGitHubController
```

Open the newly created `app/Http/Controllers/SocialGitHubController.php` file and modify it to look like this:

```
<?php

namespace App\Http\Controllers\Auth;

use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

use Auth;

use Carbon\Carbon;
use Socialite;

use App\User;

```

```
15 class SocialGitHubController extends Controller
16 {
17
18     public function redirectToProvider()
19     {
20         return Socialite::driver('github')->redirect();
21     }
22
23     public function handleProviderCallback()
24     {
25
26         $user = Socialite::driver('github')->user();
27
28         $existingUser = User::where('provider_id', $user->getId())->first();
29
30         if ($existingUser) {
31
32             Auth::login($existingUser);
33
34         } else {
35
36             $newUser = new User();
37
38             $newUser->email = $user->getEmail();
39             $newUser->provider_id = $user->getId();
40             $newUser->handle_github = $user->getNickname();
41             $newUser->password = bcrypt(uniqid());
42
43             $newUser->save();
44
45             Auth::login($newUser);
46
47         }
48
49         flash('Successfully authenticated using GitHub');
50
51         return redirect('/');
52
53     }
54
55 }
```

The `redirectToProvider` method redirects users to the OAuth service provider's authentication

endpoint. Socialite supports GitHub, so just supplying `github` to `Socialite::driver` is suffice to complete this task.

The `handleProviderCallback` method is a tad more involved. When the user is returned to the client site following provider authentication, certain additional steps will need to be completed. In this case, we're going to login the user locally if a `users` table record's `provider_id` value matches that returned by the OAuth provider. Otherwise, a new user record is created and the user is signed into the application.

After saving these changes, we'll need to complete just a few more housekeeping tasks before wrapping up this feature. Begin by adding the following two routes to your `routes/web.php` file:

```
1 Route::get('auth/github', 'Auth\SocialGitHubController@redirectToProvider');
2 Route::get('auth/github/callback', 'Auth\SocialGitHubController@handleProviderCa\
3 llback');
```

Next, add a link to the `get` route within your login view:

```
1 <a href="/auth/github">Login with GitHub</a>
```

Save these changes and try logging into your application using the GitHub OAuth provider! Presuming everything goes well, you'll add a new record to the users table without ever actually having to provide authentication credentials.

Summary

User accounts undoubtedly add another level of sophistication to your application, and Laravel makes it so easy to integrate these capabilities that it almost seems a crime to not make them available!

Chapter 10. Creating a Restricted Administration Console

Many applications require a certain level of ongoing monitoring and maintenance beyond typical code-based improvements. For instance you might wish to add new event categories, monitor newly registered users for bots, or keep tabs on event creation and interaction trends. Such tasks should be securely yet conveniently accessible to designated administrators. One effective way to integrate these capabilities is via a restricted web-based administration console. In this chapter I'll show you a simple yet effective solution for creating such a console.

Identifying Administrators

There are several third-party packages one can use to add role-based permissions to a Laravel 5 application, including perhaps most notably Entrust[82]. However if your goal is to simply separate typical users from administrators, a much more simple solution is available. You'll want to add a new column to the `users` table named something like `is_admin`. This Boolean column will identify administrators by virtue of being set to `true`. Go ahead and create the migration now:

```
$ php artisan make:migration add_is_admin_to_user_table --table=users
Created Migration: 2018_01_07_004655_add_is_admin_to_user_table
```

Next, open the newly created migration file and modify the `up` and `down` methods to look like this:

```
public function up()
{
        Schema::table('users', function(Blueprint $table)
        {
        $table->boolean('is_admin')->default(false);
        });
}

public function down()
{
        Schema::table('users', function(Blueprint $table)
```

[82] https://github.com/Zizaco/entrust

```
12          {
13                  $table->dropColumn('is_admin');
14          });
15  }
```

After saving these changes, run the migration:

```
1  $ php artisan migrate
2  Migrating: 2018_01_07_004655_add_is_admin_to_user_table
3  Migrated:  2018_01_07_004655_add_is_admin_to_user_table
```

After running the migration, all existing users will have their `is_admin` column set to `false` (the default as defined in the migration). Therefore to identify one or more users as administrators you'll need to login to your database and set those users' `is_admin` columns to `true`. For instance if you're using the `mysql` client you can login to the client and run the following command:

```
1  mysql> update users set is_admin = true where email = 'wj@wjgilmore.com';
```

Creating the Administration Controllers

Next we'll create an administration controller. In reality you'll likely wind up with several controllers which are collectively identified as being administrative in nature, so you can create a convenient *route grouping* which places them all under a *route prefix* and namespace. Create your first such controller by executing the following command:

```
1  $ php artisan make:controller Admin/UsersController
2  Controller created successfully.
```

This `make:controller` command is a bit different from the others you've executed so far throughout the book because we are *prefixing* the controller name with a directory name. In doing so, a directory named `Admin` was created inside `app/Http/Controllers`, and inside `Admin` you'll find the `UsersController.php` directory. We'll use this controller to list and manage registered users. Next let's create the route grouping which identifies both the URI prefix and the namespace:

```
1  Route::group(['prefix' => 'admin', 'namespace' => 'Admin'], function()
2  {
3      Route::resource('user', 'UsersController');
4  });
```

Note the use of `Route::group`. This allows you to nest controller inside the definition block without redundantly declaring the prefix and namespace. So for instance at some time in the future you might have three or four administrative controllers. You can follow the same approach used to create the `User` controller, and then add them to `web.php` like this:

```
Route::group(['prefix' => 'admin', 'namespace' => 'Admin'], function()
{
        Route::resource('category', 'CategoriesController');
    Route::resource('list', 'ListsController');
    Route::resource('product', 'ProductsController');
    Route::resource('user', 'UsersController');
});
```

With this route definition in place, create a new directory named `admin` inside `resources/views`, and inside it create a directory named `users`. This will house the views associated with the new administrative `User` controller. Inside the `users` directory create a file named `index.blade.php` and add the following contents to it:

```
<h1>Registered Users</h1>

<ul>
@forelse ($users as $user)

        <li>{{ $user->name  }} ({{ $user->email }})</li>

@empty

        <li>No registered users</li>

@endforelse
</ul>
```

Finally, open the new `Users` controller (`app/Http/Controllers/admin/UsersController.php`) and update the `index` action to look like this:

```
public function index()
{
        $users = User::orderBy('created_at', 'desc')->get();
        return view('admin.users.index')->withUsers($users);
}
```

With these changes in place you should be able to navigate to `/admin/user` and see a bulleted list of any registered users! Of course, before deploying this to production you'll want to restrict access to only those users identified as administrators. Let's do this next.

Restricting Access to the Administration Console

We want to allow only those users identified as administrators (their `users` table record's `is_admin` column is set to `true`). You might be tempted to make this determination by embedding code such as the following into your controller actions:

```
if (Auth::user()->is_admin != true) {
        return redirect()->route('home')->withMessage('Access denied!');
}
```

Don't do this! This is a job perfectly suited for custom *middleware*. In a nutshell, *middleware* is code that can be configured to interact with, and potentially modify, your application's request/response cycle. Middleware can do much more than restrict access, so be sure to peruse the Laravel documentation and the web in general to learn more about what's possible.

Let's create a middleware which neatly packages this sort of logic, and then associate that middleware with our administrative controllers:

```
$ php artisan make:middleware AdminAuthentication
Middleware created successfully.
```

This command created a new middleware skeleton named `AdminAuthentication.php` which resides inside `app/Http/Middleware`. Open this file and update it to look like the following:

```
namespace App\Http\Middleware;

use Closure;
use Illuminate\Contracts\Auth\Guard;
use Illuminate\Http\RedirectResponse;

class AdminAuthentication {

    public function handle($request, Closure $next)
    {
        if ($request->user())
        {
            if ($request->user()->is_admin == true)
            {
                return $next($request);
            }
        }
```

```
18
19          return new RedirectResponse(url('/'));
20
21      }
22
23  }
```

Save this file and then open `App/Http/Kernel.php` and register the middleware inside the `$routeMiddleware` array:

```
1  protected $routeMiddleware = [
2      ...
3      'admin' => \App\Http\Middleware\AdminAuthentication::class,
4  ];
```

With the middleware registered, all that remains is to associate the middleware with the route group:

```
1  Route::group(
2          [
3            'prefix'     => 'admin',
4            'namespace'  => 'admin',
5            'middleware' => 'admin'
6          ], function()
7          {
8              Route::resource('users', 'UsersController');
9          });
```

Once you've saved this change to the `routes/web.php` file, your administrative controllers will be restricted to administrators!

Summary

Hopefully this brief chapter adequately demonstrated just how easy it is to create a restricted administrative console for your Laravel applications. If you require more sophisticated role-based features then definitely check out Laravel's gates and policies features, however for more simplistic requirements I certainly suggest embracing this straightforward approach!

Chapter 11. Introducing Events and Notifications

TODO: Chapter forthcoming

Chapter 11. Introducing Vue.js

No doubt about it, JavaScript is currently the hottest, or at least the most hyped, of all programming languages. The incredibly active community has been churning out literally hundreds of new packages, libraries, and frameworks for several years now, dramatically expanding the language's applications. These days it isn't uncommon to see JavaScript being used to build desktop, mobile and server-side applications, and even within databases.

Of course, the language's original intent remains its most popular: to enhance the interactivity of web pages. Whereas in years past using the language for this purpose was generally a pretty messy process, a fair bit of order has been brought about thanks to the emergence of Node (a JavaScript run-time environment), npm (a JavaScript package manager), Webpack (a solution for sanely managing your JavaScript code within multiple files), and a wide array of libraries and frameworks intended to streamline and formalize the development of web-based JavaScript-driven applications. Even if you're a relative newcomer to JavaScript, chances are you've heard of some of these: React, Vue.js, AngularJS, and Ember.js all come to mind. These solutions bring sanity to JavaScript development, much as Composer, Laravel, and PHPUnit bring order to yours.

Laravel developers have been particularly drawn to Vue.js[83] (heretofore referred to as just *Vue*), an open source framework created by former Google developer Evan You. I'd posit the Laravel community's attraction to Vue is due in part to it being the first such JavaScript framework incorporated into Laravel by default, and in part because it espouses a no-nonsense approach to developing web applications which in many ways echoes Laravel's own practical perspective. In this chapter you'll learn all about Vue as we update HackerPair to include several new dynamic features.

This chapter offers a gentle introduction to Vue, but it is by no means comprehensive. My goal with this updated release is to provide you with enough information to begin integrating basic Vue-driven features into your own Laravel applications with as little frustration as possible. Be sure to consult the fantastic Vue documentation and associated tutorials for a complete introduction!

Installing Vue

Like most popular JavaScript libraries, Vue can be installed in several different ways. However, if you're new to the world of modern JavaScript development, configuring the environment and its typically many dependencies can be an incredibly intimidating task. Fortunately, the Laravel

[83] https://vuejs.org/

developers have taken care of the vast majority of the installation- and configuration-related issues for you, allowing you to begin experimenting with Vue code almost immediately.

In chapter 2 you learned how about Laravel Mix, which uses Node and npm to create a convenient solution for carrying out otherwise tedious tasks such as CSS and JavaScript compilation. To initialize Mix you need to run the `npm install` command, which installed the Node packages identified in the `package.json` file. If you open `package.json` you'll see the Vue package is included in this list. Therefore if you haven't yet run `npm install` navigate to your project's home directory and do so now:

```
$ npm install
```

Next you should install the vue-devtools[84] browser extension. Extensions are available for Chrome, Firefox, and Safari, so you'll definitely want to use one of these browsers for Vue-related development. You can install the appropriate extension by heading over to the vue-devtools GitHub page and navigating to your browser's respective extension download link. We'll return to this extension throughout the chapter, so be sure to install it before moving on.

With that done, let's have a look at `resources/assets/js/app.js`:

```
require('./bootstrap');

window.Vue = require('vue');

Vue.component('example-component',
  require('./components/ExampleComponent.vue'));

const app = new Vue({
    el: '#app'
});
```

This JavaScript file is responsible several key tasks:

- Loads the `bootstrap.js` module (also found in `resources/assets/js`). This file is responsible for loading jQuery, the Bootstrap jQuery plugins, ensuring Laravel's CSRF token is passed as a header when making AJAX requests (more about this later in the chapter), and optionally loading Laravel's Echo API.
- Loads the Vue module, which we'll obviously need for integrating Vue features into the application.

[84] https://github.com/vuejs/vue-devtools

- Loads an example Vue component creatively named `ExampleComponent.vue`. This component is found in the `resources/assets/js/components` directory. We'll return to this file in a moment, but if you're new to Vue just keep in mind that a Vue component allow you to create reusable HTML widgets which are typically enhanced with dynamic, JavaScript-driven behavior. In this chapter we'll create multiple Vue components while exploring different Vue capabilities.
- Creates a new Vue instance, and identifies the *root element* of the Vue application. This root element serves as a cue to Vue that all Vue-related output will be rendered somewhere inside this element. This is often done simply by wrapping everything within your layout `<body>` with a `<div id="app">...</div>` element. As you'll soon see we'll rely on precisely this approach.

Let's turn our attention to the aforementioned `ExampleComponent.vue` file.

Creating Your First Component

Laravel includes a simple Vue component intended to acquaint you with the basic steps required to integrate a component into your application. Although simplistic, it does serve to familiarize you with how a typical component is structured. I'll present the example component's code next, followed by some commentary:

```
<template>
    <div class="container">
        <div class="row">
            <div class="col-md-8 col-md-offset-2">
                <div class="panel panel-default">
                    <div class="panel-heading">Example Component</div>

                    <div class="panel-body">
                        I'm an example component!
                    </div>
                </div>
            </div>
        </div>
    </div>
</template>

<script>
    export default {
        mounted() {
            console.log('Component mounted.')
```

```
        }
    }
</script>
```

This file is broken into two primary sections. The section encapsulated by the `<template>` tag defines the widget's HTML. It references various Bootstrap 3 classes, but even if you're not using Bootstrap 3 the HTML will still render once the component is incorporated into your application (albeit not as the Bootstrap 3 developers intend it to). However for the purposes of illustration let's simplify the template to look like this:

```
<template>
    <div>
        I'm an example component!
    </div>
</template>
```

The section encapsulated by the `<script>` tag defines the component's logic. This example uses one of Vue's lifecycle hooks (`mounted()`) to write a message to the browser console once the component has been rendered to the page DOM. Leave this code untouched for now; we'll modify it plenty in forthcoming examples.

Even novice JavaScript developers are probably wondering how the `ExampleComponent.vue` file is going to be executed in the browser since it is not valid JavaScript. We're going to use Laravel Mix to compile the Vue code into standard JavaScript! Open a terminal and execute the following command:

```
$ npm run dev
```

This will compile all of the modules and other JavaScript code imported by the `app.js` file, and save the results to `/public/js/app.js`. Next we'll want to integrate this file, the designated Vue root element, and the Vue component into our application. In order to most effectively demonstrate this I'm going to recommend creating a new layout and view. Create a new layout named `vue.blade.php`, placing it in your project's `resources/views/layouts` directory. It should look like this:

```
<!DOCTYPE html>
<html>
<head>
<title>Vue Examples</title>
</head>
<body>
    <div id="app">
            @yield("content")
    </div>
        <script src="/js/app.js"></script>
        </body>
</html>
```

This layout satisfies two key requirements:

- It defines the Vue root element (`#app`). Any referenced Vue components must be placed inside this element. We'll reference the example component inside a view which we'll create in just a moment.
- It references the generated `app.js` file. It's standard practice is to do so right before the closing `</body>` tag.

Next, create a new view named `vue.blade.php`, placing it in your `resources/views` directory. Add the following to this file:

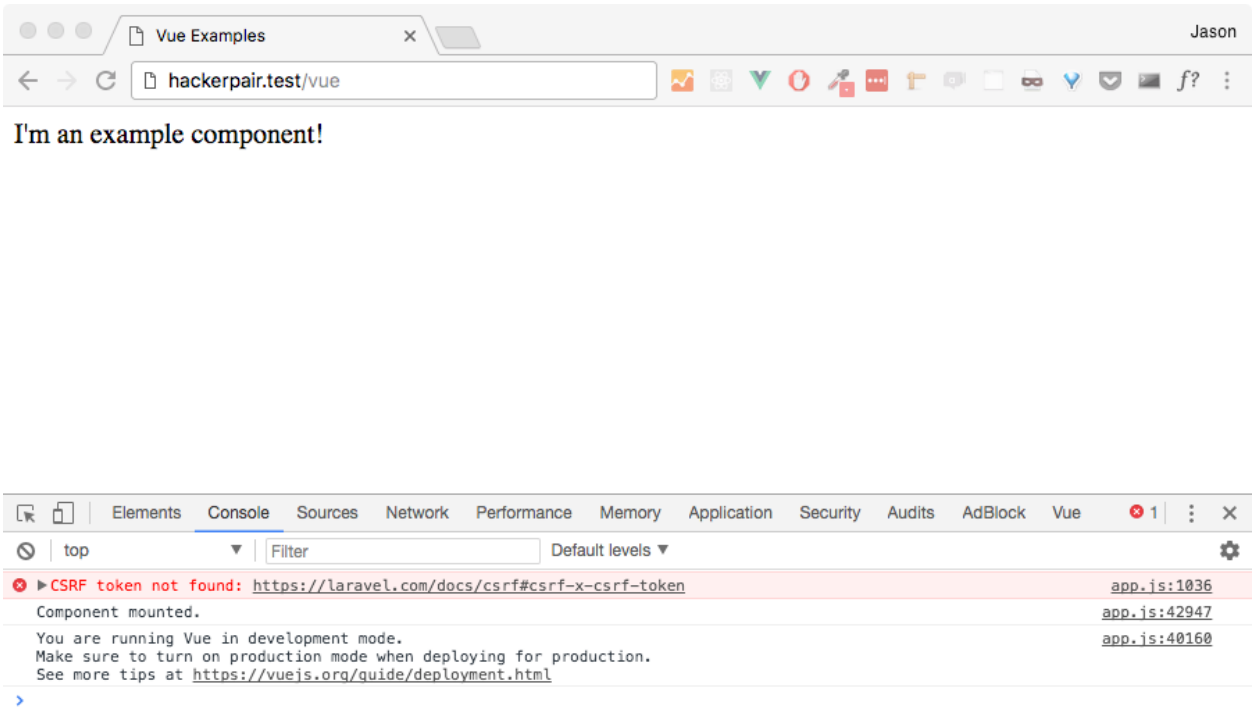
```
@extends('layouts.vue')

@section('content')
        <example-component></example-component>
@endsection
```

That's right, you can reference the Vue component just like an HTML tag! If you return to `app.js`, notice how we've aliased the `ExampleComponent.vue` file to the name `example-component`. The matter of naming in Vue practically warrants a section unto its own, however I'm going to boil the matter down to two crucial rules when it comes to components. First, you should always use compound names when naming components, to prevent conflicts with current and forthcoming HTML elements. This is because HTML elements cannot be named using a compound word. Second, components should always be Pascal-cased (e.g. `ExampleComponent.vue` as opposed to `exampleComponent.vue` or `examplecomponent.vue`) or kebab-cased (e.g. `example-component.vue`). I'll return to other naming convention tips as warranted throughout the chapter.

Save these changes, and then add the following line to your `routes/web.php` file:

```
1  Route::view('vue', 'vue');
```

With these pieces in place, open the browser and navigate to your project's `/vue` URI and you should see output similar to that presented in the following screenshot:



Rendering your first Vue component

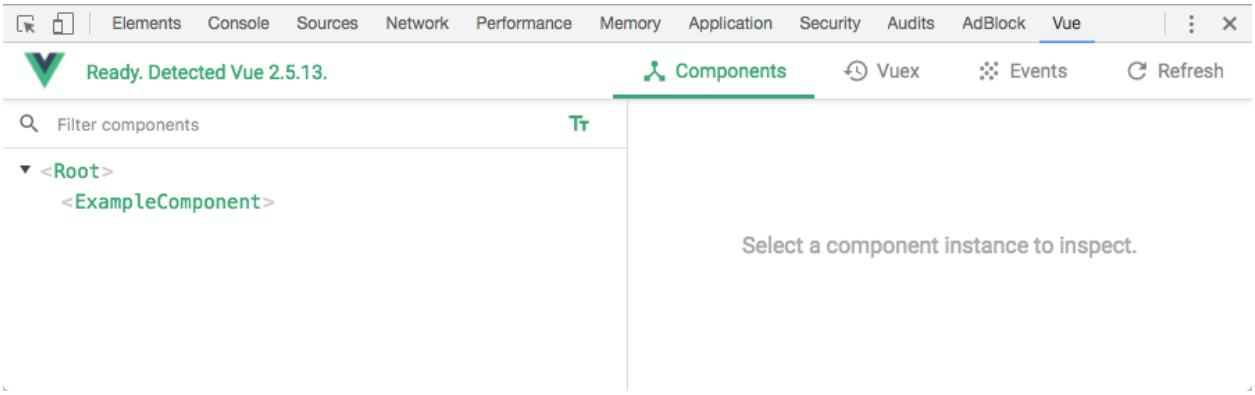
Notice in this screenshot I've opened the Chrome development console, and inside it are three noteworthy messages:

- *CSRF token not found*: As a safeguard against cross-site forgery attacks, Laravel automatically generates a token for each of your application's user sessions. This token is repeatedly passed between the client and server to confirm the correct user is performing application requests. However, when using AJAX-based requests this token must be passed via the request header. If you return to `bootstrap.js` you'll see the Axios library (introduced later in this chapter) looks for this token inside an HTML meta tag. You can resolve this error by adding this line inside your application layout's `<head>` tags:

```
1     <meta name="csrf-token" content="{{ csrf_token() }}">
```

- *Component mounted*: Recall the `ExampleComponent.vue` file logs this message to the console once the component has been mounted. This is not a requirement, and just serves to demonstrate both Vue's lifecycle hooks and the ability to send messages to the console.
- *You are running Vue in development mode*: Your Laravel project's JavaScript build environment is able to determine when your project is running in development mode, allowing Vue to display useful information about error and syntax within the browser's console. Therefore it's a good idea to keep your console open while working on new Vue features.

While you're still in the console, click on the console's `Vue` tab (presuming you followed my advice and installed the vue-devtools extension). You should see an interface which looks like that presented in the below screenshot:



The vue-devtools browser console tab

This extension can be incredibly useful for debugging Vue components, particularly when you begin nesting them later in the chapter. In this screenshot you can see the `ExampleComponent` nested under the `Root` component. Click the `ExampleComponent` reference and you'll see the message `This instance has no reactive state`. That will change in the next section.

Integrating Reactive Data Into Your Component

The last example served to explain how a Vue component is integrated into your application, but because the component displayed static data there really isn't much of a need for a component, is there? Let's ease into integrating dynamic data into your component by introducing the concept of *reactivity*. Vue is *reactive*, meaning you can link component data to the DOM in such a way that if the data is changed it will automatically be rendered anew to the page (without requiring a page reload). To demonstrate this behavior, return to the `Example` component and modify it to look like this:

```
<template>
    <div>
        {{ greeting }}
    </div>
</template>

<script>
    export default {
        data() {
            return {
                greeting: "I'm an example component!"
            }
        },
        mounted() {
            console.log('Component mounted.')
        }
    }
</script>
```

Save these changes and recompile the source files:

```
$ npm run dev
```

Incidentally, you can save a bit of hassle when repeatedly modifying Vue components by instead running `npm run watch`. Doing so will result in your files automatically being recompiled every time modifications are saved.

With the changes in place, reload your browser and you'll see... the same outcome as the last example. While this seems anticlimactic, the fact the message was rendered by way of a property found in the data object is actually indicative of an important change in behavior. To prove it, open your browser console and execute the following:

```
$vm.greeting = "Vue is reactive!"
```

Upon executing this statement, the greeting will *automatically* change from `I'm an example component!` to `Vue is reactive!`. The `$vm` is just a variable the Vue developers chose to represent the Vue instance. Of course, you won't typically interact with your Vue components via the console, but it does serve as a useful tool for demonstration. The important idea to keep in mind here is that your underlying Vue code *can* change this data after the page has been rendered, whether it be due to a user-initiated event such as a click, or ongoing updates to data returned from the database.

You can accomplish the same outcome by instead clicking on the console's `Vue` tab (from here on out I'll just presume you've installed it). Then click on the `ExampleComponent` reference, and instead of

seeing the message `This instance has no reactive state`, you'll see the object returned by the `data` function and the associated `greeting` property. Mouse over the property value and you'll see a pencil. Click that pencil and you'll be able to change the value, but make sure the string remains encapsulated by quotations.

Let's try one more variation of this example before moving on. Return to the `Example` component and assign the `greeting` property the following value:

```
greeting: new Date().toLocaleString()
```

Save these changes and run `npm run dev` again (or just return to your browser if you're already running `npm run watch`). Reload the browser and you'll see your local date and time. Reload again and you'll see this value change accordingly.

Responding to Browser Events

While the previous example demonstrated the ability to render a JavaScript-generated dynamic value (the local date and time), you had to actually reload the browser in order to see that value change. We could accomplish the same thing using PHP! Let's create a more compelling example by updating that date and time value every time a button is clicked. Return to the `Example` component and modify it to look like this:

```
<template>
  <div>
    {{ greeting }}<br />
    <button @click="updateTimestamp">Update</button>
  </div>
</template>

<script>
    export default {
    data() {
      return {
        greeting: "Click the button to view the latest date and time!"
      }
    },
    methods: {
      updateTimestamp() {
        this.greeting = new Date().toLocaleString()
      }
    }
    }
</script>
```

The `data` object's `greeting` property is rendered as before within the template, and you'll additionally see a button which includes the following declaration:

```
@click="updateTimestamp"
```

This tells Vue to bind the button's click event to a method named `updateTimestamp`. You'll see this method defined in a new literal named `methods`. Every time the `updateTimestamp` method is executed, the `greeting` property is updated to reflect the very latest local timestamp. Give it a whirl and you'll see the timestamp updated every time you click the button!

Vue supports plenty of other event handlers. Be sure to consult the documentation for a complete summary.

Working with Props

You'll often want to pass data from a parent to a child component. There are many reasons for wanting to do so, perhaps the most obvious being a desire to pass a record ID along in order to subsequently perform an AJAX request for reason of retrieving more information about that record. You'll learn about performing AJAX requests in the later section, "Integrating AJAX Requests with Axios", so for the moment let's focus on passing the data.

Let's modify the `Example` component yet again to accept a parameter. In the context of Vue, these parameters are referred to as *props*. These props are passed into the component like so:

```
<example-component greeting="I really love Vue!"></example-component>
```

Next, modify the `Example` component to look like this:

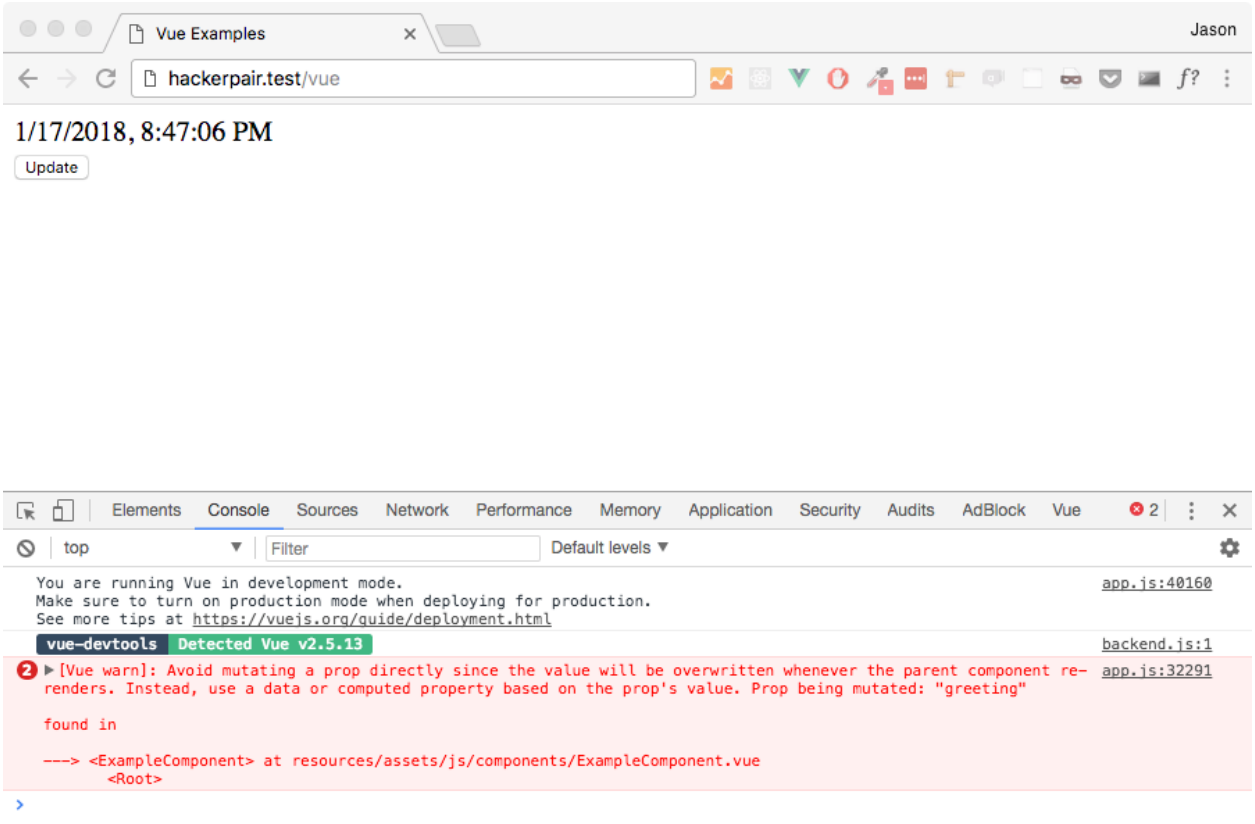
```
<template>
    <div>
        {{ greeting }}
    </div>
</template>

<script>
    export default {
        props: {
          greeting: String
        }
    }
</script>
```

The names of any supplied props must be defined within the `props` option. While it is possible to define these props in a simple array (e.g. `['greeting']`), best practice dictates at a minimum you should additionally specify the prop type (other specifications are also supported, such as defining whether the prop is required). Once defined, you're free to use them within your component, including within the component template as demonstrated above.

Keep in mind you're never supposed to modify a props value within the child component. Once of the central tenets of Vue is the idea of *one-way data flow*. This means any data flowing from the parent to the child should *never* be modified, lest the modification inadvertently modify the parent's state. To demonstrate this restriction, modify the `Example` component to look like this:

```
<template>
    <div>
        {{ greeting }}<br />
        <button @click="updateTimestamp">Update</button>
    </div>
</template>

<script>
    export default {
        props: {
          greeting: String
        },
        methods: {
          updateTimestamp() {
            this.greeting = 'HackerPair'
          }
        }
    }
</script>
```

Reload these changes and click the `Update` button. In doing so your component will modify the `greeting` prop. Vue will let you do so, however the console will render the warning presented in the following screenshot.



Never modify Vue props

We'll return to props later in this chapter when introducing some of the Vue-driven features incorporated into the HackerPair companion project.

Rendering Lists

Whether its events, locations, or categories, much of the HackerPair application is involved in the business of merely rendering lists of data to the screen. Even if your application were entirely Vue-driven, much of the logic and presentation would be tasked with carrying out this fundamental role. So it should come as no surprise that list rendering is native to Vue. In this section we'll continue progressing towards the goal of showing you how to use Vue in conjunction with Laravel to render lists of events using AJAX. For the moment we'll just hard-code an array of event titles within a component, and focusing on the matter of iterating over this array to render the list.

To demonstrate this capability, let's create a new component named `EventList`. Create a new file named `EventList.vue`, and place it inside the `components` directory. Add the following contents to it:

```
<template>
    <div>
        <h1>HackerPair Events</h1>
        <ul>
            <li v-for="event in events">
                {{ event }}
            </li>
        </ul>
    </div>
</template>

<script>
    export default {
        data() {
            return {
                events: [
                    'Laravel and Coffee',
                    'The Wonders of IoT',
                    'All About ES6',
                    'Introducing Vue'
                ]
            }
        }
    }
</script>
```

This component is no structurally different from the `Example` component in that it consists of a template and associated logic. The `data` function returns an array of event names via the `events` property. This array is iterated over using the `v-for` directive. Notice how this directive is used as an argument to the `li` element. This loops over each item in the `events` array, rendering a new `li` element and the current `events` item with each iteration.

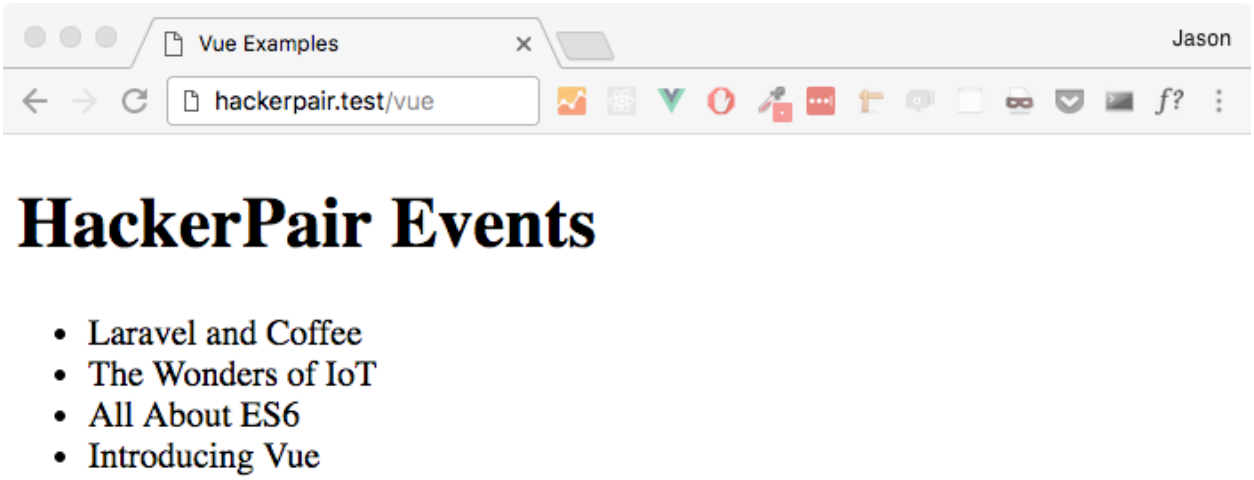
In order to use this component, you'll need to register it within the `app.js` file, just as was done with the `Example` component.

```
Vue.component('event-list',
    require('./components/EventsList.vue'));
```

With this in place, return to the `vue.blade.php` view and swap out the `example-component` element with `event-list`:

```
<event-list></event-list>
```

Finally, reload the browser and you should see the event list rendered just as is depicted in the below screenshot.



Iterating over events using v-for

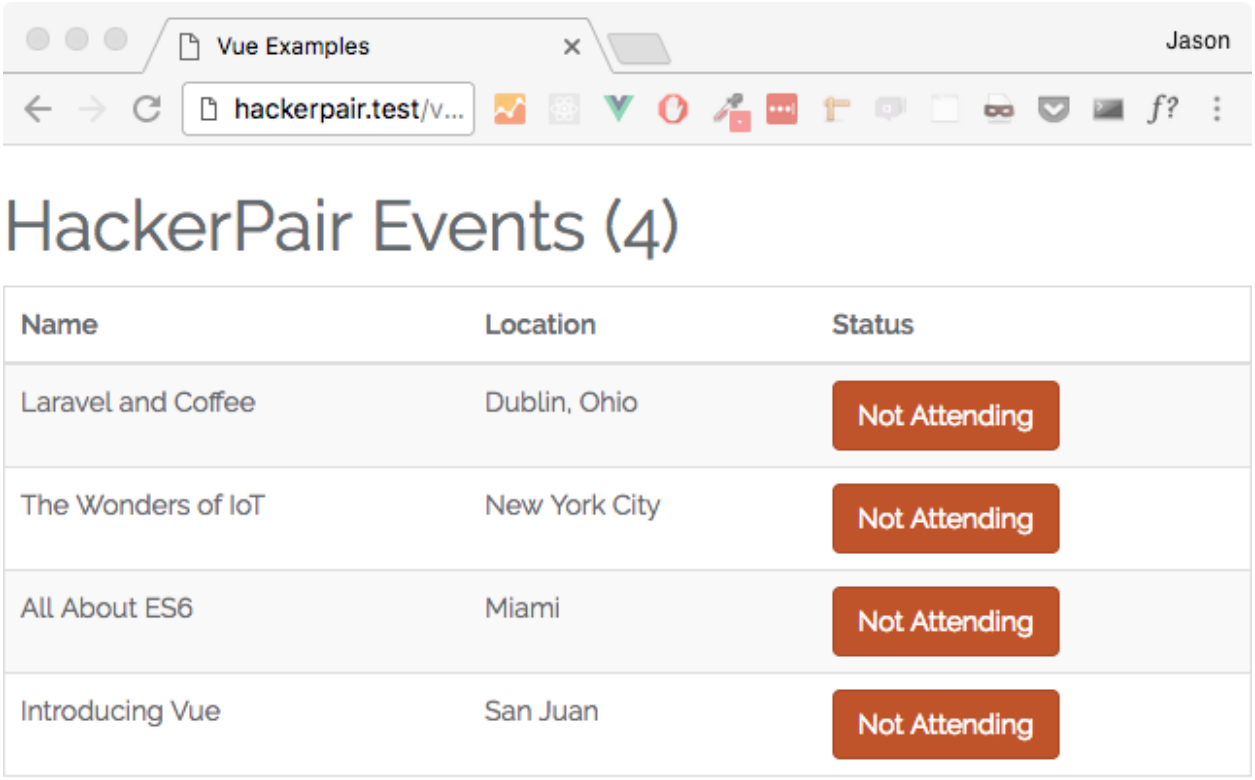
This is but the simplest demonstration of `v-for`. Be sure to consult the Vue documentation for more examples. Further, Vue also supports directives for conditionally rendering data, the most notable solution being the `v-if` directive. Check out the Vue documentation for more information about conditional rendering, and I'll be sure to add a section on the topic in a forthcoming release.

Nesting Components

So far I've been introducing various Vue features alongside very simple examples. This is by design; attempting to jump straight into more involved examples is a surefire way to become frustrated and

quite possibly quit learning before you've even started. However at this point in the chapter you've learned enough fundamentals to begin considering more realistic examples. The most logical first step in this direction involves understanding how to *nest components*.

Returning to the event list created in the last section, the interface was very simple, consisting of a straightforward unordered list. But what if the interface were more complicated, such as that presented in the following screenshot.



A more complicated event listing

While you certainly could pack all of the markup and logic into a single component, your sanity will be much better served by breaking a layout such as this into two or even more components. This way you can manage the layout using much more manageable templates, and neatly encapsulate the associated logic for each. Let's start with the parent component. We'll continue using the `EventList` component created in the last section, which I'll recreate here for easy reference:

```
<template>
    <div>
        <h1>HackerPair Events {{ events.length }}</h1>
        <ul>
            <li v-for="event in events">
                {{ event }}
            </li>
        </ul>
    </div>
</template>

<script>
    export default {
        data() {
            return {
                events: [
                    'Laravel and Coffee',
                    'The Wonders of IoT',
                    'All About ES6',
                    'Introducing Vue'
                ]
            }
        }
    }
</script>
```

Instead of using a simple unordered list, we're going to iterate over a child component, passing information about each event along as a prop. Let's begin by modifying the `EventList` template:

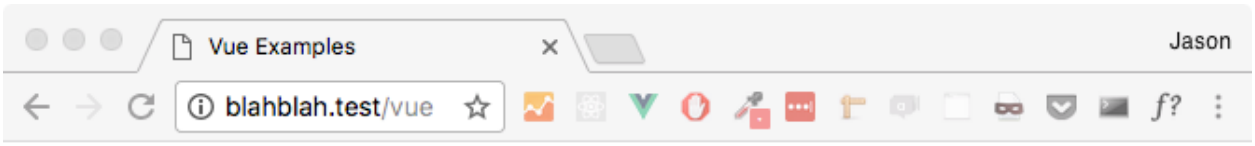
```
<template>
    <div>
        <h1>HackerPair Events ({{ events.length }})</h1>
        <table class="table table-striped table-responsive">
            <thead>
                <tr>
                    <th>Name</th>
                    <th>Location</th>
                    <th>Status</th>
                </tr>
            </thead>
            <tbody>
                <event-item
```

```
                    v-for="event in events"
                    :event="event"
                    :key="event.id">
                </event-item>
            </tbody>
        </table>
    </div>
</template>
```

This time I'm using Bootstrap in order to achieve the layout presented in the previous screenshot. That's fairly standard though; what is important is the inclusion of the `event-item` component:

```
<event-item
    v-for="event in events"
    :event="event"
    :key="event.id">
</event-item>
```

We're using the `v-for` directive just as we did in the previous section, but this time have embedded it within a child component reference rather than an `li` element. As before, each `event` peeled from the `event` array is expressly identified as the prop value `event`, and a second attribute named `key` is additionally assigned the same prop value. I'll talk more about the purpose of `key` in a moment but right now I want you to pay particularly close attention to how `key` and `event` are defined. Both are prefixed with a `:`. This is because these values are being dynamically assigned by the `v-for` looping mechanism. If you neglect to prefix `key` and `event` with a colon, Vue will presume you meant to pass the static string `name` along, and produce output like that found in the below screenshot.



HackerPair Events (4)

- name
- name
- name
- name

Mistakenly displaying static strings

Returning to why the `key` attribute is included, it's commonplace for Vue-driven features to allow for the insertion, deletion, update, and rearrangement of list items such as these events. Associating a unique key with each item allows Vue to keep track of each unique item while making requisite changes to the DOM. So unless you're implementing a simple interface which is effectively static beyond the initial rendering, you'll want to include keys. In a typical implementation you'll assign something definitively unique such as a database record ID, and as you can see I'm assigning `event.id` to the key. But wait, this looks like an object rather than a string-based array element. Where did this come from? In an effort to bring us ever closer to the goal of retrieving an array of (JSON) objects from the database using AJAX, I've modified the `events` array to mimic this data structure. It's found in the updated `EventList` component logic:

```
<script>
    import EventItem from './EventItem'

    export default {
        components: {
            'event-item': EventItem
        },
        data() {
          return {
            events: [
              { id: 1,
                name: 'Laravel and Coffee',
                location: 'Dublin, Ohio'
              },
              { id: 2,
                name: 'The Wonders of IoT',
                location: 'New York City'
              },
              { id: 3,
                name: 'All About ES6',
                location: 'Miami'
              },
              { id: 4,
                name: 'Introducing Vue',
                location: 'San Juan'
              }
            ]
          }
        }
    }
</script>
```

Because use of the `EventItem` is (presently) limited to use in conjunction with the parent `EventList` component, we're importing it expressly into the `EventList` component rather than doing so globally within the `app.js` file. After doing so, we can define the component alias (`event-item`) within `components`. Next, you'll see the `events` array has been modified to contain an array of objects, which is very similar to what will be returned once we start retrieving this information using AJAX. Each event includes an ID, name, and location. When using `v-for` to iterate over each event, that event object is passed along as a prop to the `EventItem` component. Let's create this component next.

Create a new file named `EventItem.vue` and place it in your project's `components` directory. Add the following contents to it:

```
<template>
    <tr>
        <td>
            {{ event.name }}
        </td>
        <td>
            {{ event.location }}
        </td>
        <td>
            <button class="btn btn-danger">Not Attending</button>
        </td>
    </tr>
</template>

<script>
    export default {
        props: {
            event: Object
        }
    }
</script>
```

Each time this component is rendered within `EventList`, a new row will be added to the table, containing the provided event's name and location. Note we're not required to manually manage the key; this is something Vue handles autonomously.

The sky is the limit from here. You could for instance bind an event handler to the button which when clicked updates the user's attendance status in conjunction with this event. This will involve performing an AJAX request, which is covered in the next section, but in the meantime you could mimic the outcome by adding the event handler and an associated method which toggles the button's style between `btn-danger` and `btn-success`, thereby updating the button color. You can learn all about updating element styles using Vue in the Vue documentation.

Now that you've seen how an static array of event objects can be iterated over and passed along to a child component, I'd bet you're particularly excited about taking the next logical step and retrieving this event information from the database using AJAX. We'll tackle this topic next.

Integrating AJAX Requests with Axios

If you return to `package.json` (in your project's root directory) you'll see a package named Axios[85] has been installed. In the context of Vue, Axios greatly simplifies the task of performing AJAX

[85] https://github.com/axios/axios

requests. Having in the past worked on a great many of projects which integrate AJAX calls, Axios has been a breath of fresh air thanks to it's intuitive syntax and use of the Promise API[86].

I'm unavoidably referring to some intermediate JavaScript concepts in this section, including features such as "promises". It's difficult to know where to draw the lines in terms of introducing fundamental concepts as a precursor to presenting these sorts of examples without radically expanding the chapter, so at least for this release I'll refer you to the excellent Mozilla Developer Network's JavaScript Guide[87] for more information about these crucial JavaScript features.

In the last section you learned how to use nested components to create a maintainable event listing interface. The list of events was managed as an array of objects within the parent `EventList` component, and each array element was passed along to the child `EventItem` component as a prop. We'll revise the `EventList` component to instead query the database using AJAX, but first let's create a new controller which will respond to AJAX calls. Unlike the `Events` controller created in chapter 3, this controller will not return a view. Instead it will return data in JSON format. I'll talk at length about creating an API in chapter 13 but let's get a little head start now and create a new directory which will house that API and various controllers:

```
$ php artisan make:controller API/EventsController
```

This created a new directory inside `app/Http/Controllers` named `API`. Inside that directory is the newly created controller (`EventsController.php`). Open this file and add the following method:

```
public function index()
{

    return Event::orderBy('start_date', 'desc')->take(10)->get();

}
```

I'm only retrieving 10 records rather than using the `all` method because my HackerPair development database's `events` table contains dozens of records and I don't want to load them all into the table. We'll change this soon when later in the section I show you how to paginate records in Vue. Next, add the following route to your `web.php` file:

```
Route::resource('/api/events', 'API\EventsController');
```

[86] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises
[87] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide

With the new controller, action, and route in place, test it out by navigating to the `/api/events` URI in your browser. You should see a JSON-formatted response containing 10 event records. Note we did not have to include any JSON-specific logic; Laravel converts a collection into JSON when the collection is explicitly returned from an action.

Next, let's revise the `EventList` component to instead obtain this list of events from the HackerPair database. Open `EventList.vue` and replace the `events` array with:

```
error: '',
events: []
```

We'll use the new `error` property to display an error should anything but a `200` HTTP status code be returned from the AJAX call. Next, add the following `created` function and a `methods` object literal which defines the `getEvents` function:

```
created() {
    this.getEvents()
},
methods: {
    getEvents() {

        axios.get('/api/events')
            .then(response => {
                this.events = response.data
            })
            .catch(e => {
                this.error = "An error has occurred."
            })

    }
}
```

The `created` function is a lifecycle hook like `mounted`. It will execute before the template has been rendered, but has access to component data and events. It is ideal for performing AJAX requests because we can retrieve the data, populate the `events` array, and render it within the component template before that template is rendered to the browser window.

Should a non-200 HTTP status code be returned (e.g. a 404 or 500), Axios will throw an exception, causing the code inside the `catch` block to be executed. In the above example a simple error message is assigned to the `error` property. You can then display this property within the template by adding it to your component template as demonstrated below:

```
<template>
    <div>
        <h1>HackerPair Events ({{ events.length }})</h1>
        {{ error }}
        ...
```

You can test this by modifying the new `Events` controller's `index` action to return a `500` status code instead of a `200`:

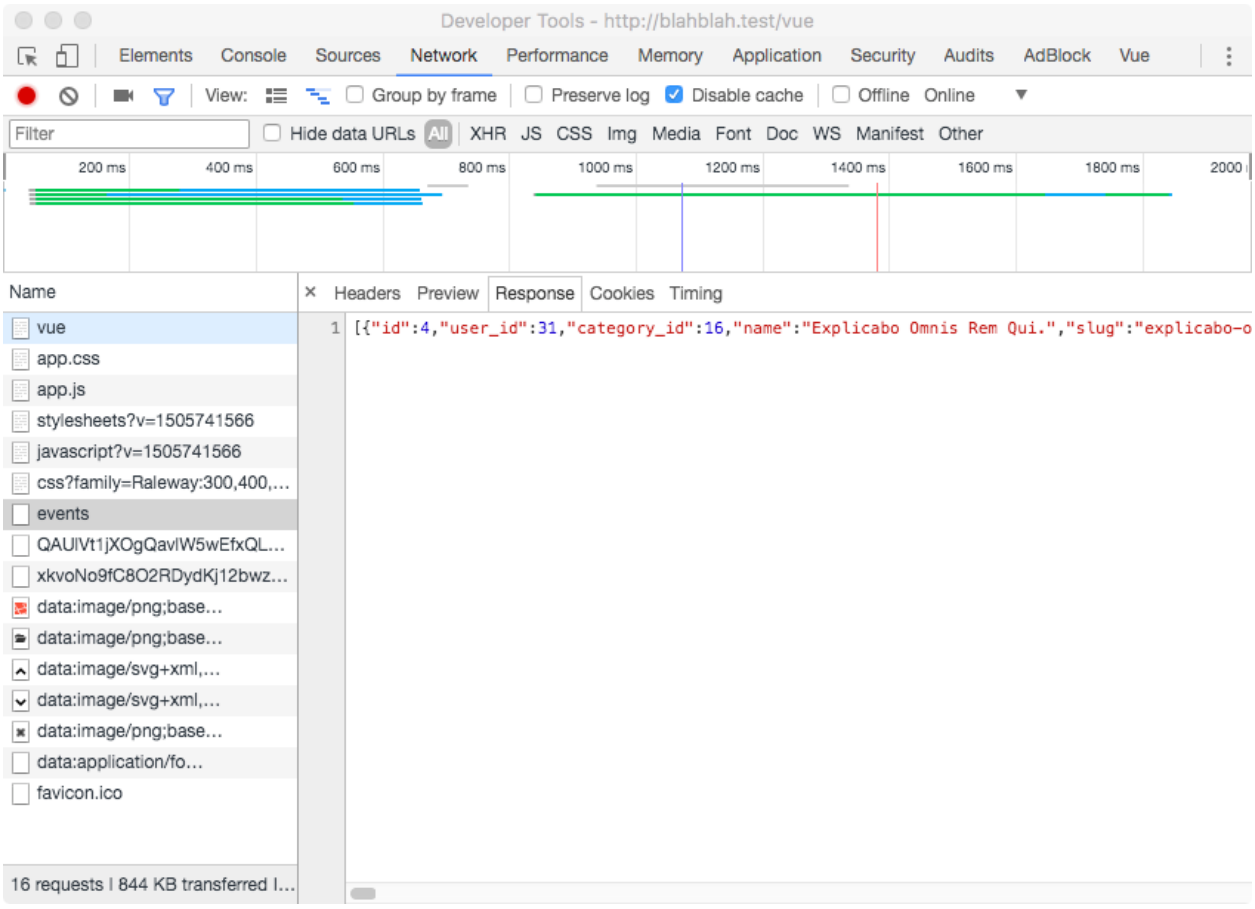
```
return response()->json(['status' => 'error'], 500);
```

Of course, there's plenty of room to carry out more sophisticated (and eye-appealing) responses should an error occur, such as passing the error to a notifications library such as vue-notification[88].

Filtering Unwanted Attributes

Keep in mind *the entire* AJAX response can be viewed by any user via the browser console. For instance if you open up Google's Developer Tools console and click the Network tab, you'll see the entire JSON response by selecting `events` in the left-hand menu and then clicking the `Response` tab. This process is illustrated in the below screenshot:

[88] https://github.com/euvl/vue-notification



Viewing AJAX responses in Chrome

Chances are you'll want to ensure some record attributes are not returned. You can do this in two ways. The first involves adding the `select` method to your query in order to restrict what's returned:

```
return Event::select('name', 'city')->orderBy('start_date', 'desc')
    ->take(10)->get();
```

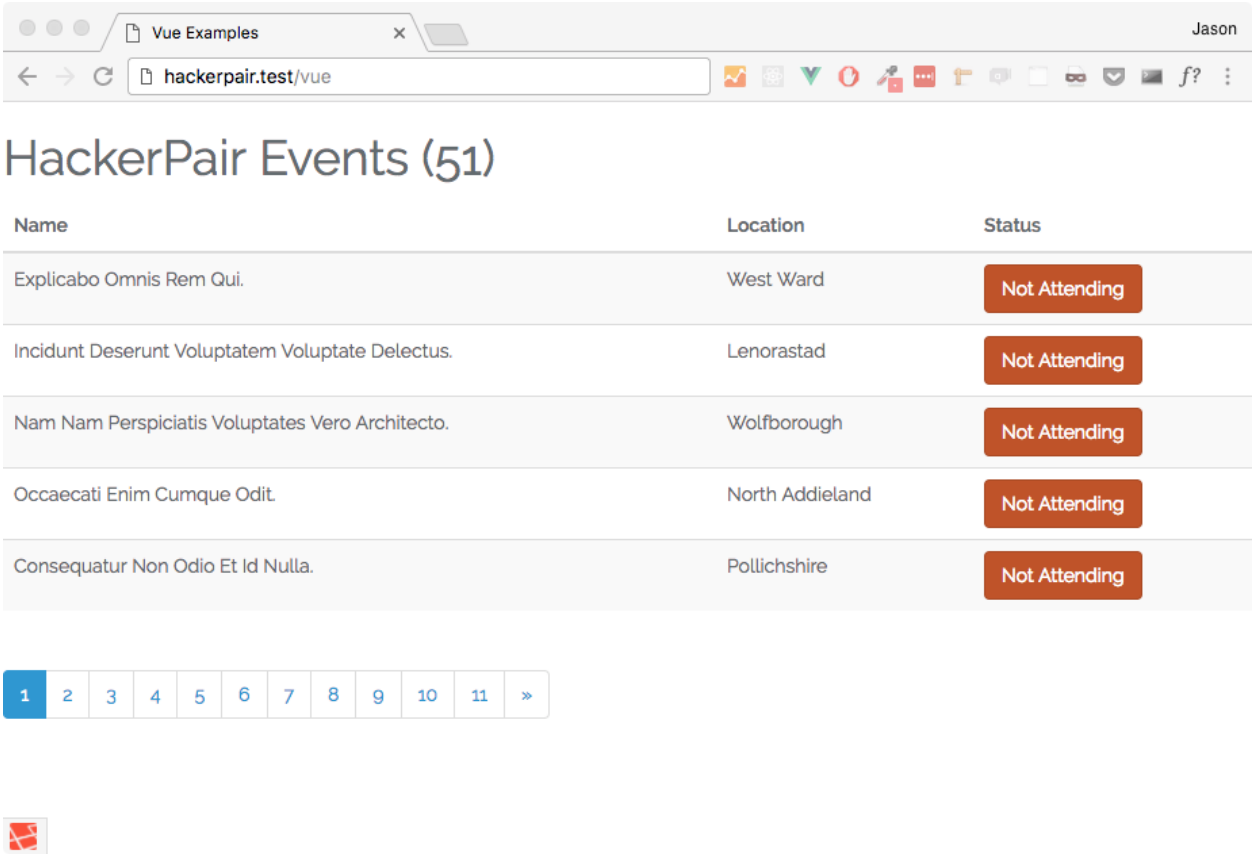
Alternatively, you can use the `$hidden` property in your model to always filter out certain attributes:

```
class Event extends Model
{

    $hidden = ['published']

    ...

}
```

Integrating Pagination

The `Events` controller's `index` action currently returns just ten entries in order to avoid dumping all available records to the component table. This isn't going to be a practical solution for the production site, so let's make the modifications necessary to paginate the results. Unfortunately, this isn't as easy as merely enabling Laravel's native pagination feature (introduced in chapter 3), but it does present an opportunity to learn something new so let's dive right in.

Our finished page is presented in the following screenshot.



Paginating database results with Vue

Begin by installing the Laravel Vue Pagination package. Open a terminal and navigate to your project's root directory, then execute this command:

```
$ npm install laravel-vue-pagination
```

Keep in mind there are plenty of packages available for paginating Laravel results with Vue, however this one is compatible with Bootstrap 3 and 4. Be sure to look around for other solutions before settling on one.

Next, load the component into your application. Because it's likely you'll be using pagination throughout your application, load the component via the `app.js` file:

```
Vue.component('pagination', require('laravel-vue-pagination'));
```

Next, we'll need to modify the `EventList` component's `getEvents` method and template. Laravel's paginator needs to be presented with the current page every time it's called, so we need to pass a `page` parameter into `getEvents`, which is subsequently passed along to the `/api/events` URI. Because the `page` parameter doesn't exist on the initial page load, we'll need to set it to `1`:

```
getEvents(page) {

    if (typeof page === 'undefined') {
        page = 1
    }

    axios.get('/api/events?page=' + page)
        .then(response => {
            this.events = response.data
        })
        .catch(e => {

            this.error = "An error has occurred."
        })

}
```

Next we need to modify the template. Laravel's paginated data structure is different from that returned when using for instance `all`, because additional information is passed along. The data structure looks like this:

```
{
    current_page: 1,
    data: [{name: 'Laravel and Coffee', 'city' => 'Dublin'}, ...],
    from: 1,
    last_page: 10,
    next_page_url: 'http://hackerpair.com/api/events?page=2',
    per_page: 15,
    prev_page_url: null,
    to: 15,
    total: 51
}
```

Therefore the `v-for` directive must be provided with `events.data` in order to loop over the records. However the `pagination` component requires the entire data structure in order to present the pagination ribbon. Additionally, it needs to know which method to execute when the user clicks on one of the numbered panels in the ribbon. Below you'll find the `EventList` component's template in its entirety:

```
<template>
    <div>
        <h1>HackerPair Events ({{ events.total }})</h1>
        {{ error }}
        <table class="table table-striped table-responsive">
            <thead>
                <tr>
                    <th>Name</th>
                    <th>Location</th>
                    <th>Status</th>
                </tr>
            </thead>
            <tbody>
                <event-item
                    v-for="event in events.data"
                    :event="event"
                    :key="event.id">
                </event-item>
            </tbody>
        </table>

        <pagination :data="events"
          v-on:pagination-change-page="getEvents"></pagination>

```

```
25      </div>
26  </template>
```

Summary

Vue.js offers an incredibly powerful solution for taking your application's user interface to the next level, yet in my experience allows you to do so by investing a fraction of the time and effort required of competing solutions.

Chapter 12. Creating an Application API with Laravel Passport

In the early days of the web, a project website was all-important, with every effort put into driving traffic towards it. This metric was historically considered so important that startups formed during the dot com bubble of the late 1990's were valued more according to their ability to drive visitors (nauseatingly known as "eyeballs" at the time) than for their ability to make money.

Over time though, companies came to realize it wasn't the website that offered value but rather the underlying data. For instance Google Maps' official website (https://www.google.com/maps) is a technological marvel however it didn't take Google long to realize the real star was the trove of location-based data it had amassed through satellite technology and sophisticated programming. To capitalize on this data, Google released its first mapping API just a few months after the website launched. This resulted in an explosion of mapping applications and set the stage for Google's dominance in consumer mapping technology for years to come. Incidentally, Google's not alone in their penchant for providing API-driven services. These days most major online services offer APIs, among them Amazon, Facebook, GitHub, and Twitter.

Chances are your own project could benefit from offering an API. Properly implemented and secured, your users may very well devise new ideas for your data which had never previously occurred to you. Even if you didn't open up the API to others, you could use the API to build for instance a native iOS or Android application. In this chapter I'll introduce you to REST-based API development, showing you how to add an API to the HackerPair application.

This chapter should provide enough information to get you started, but if you're interested in gaining a truly well-rounded understanding of API development, pick up a copy of Phil Sturgeon's book, "Build APIs You Won't Hate". Learn more at https://apisyouwonthate.com/[89].

API Fundamentals

Despite your involvement in web development, it's entirely possible you haven't had the opportunity to use a third-party API, let alone build one. So before jumping into building an example HackerPair API, let's kick things off by talking in general terms about how an API works. An API, or *application program interface*, provides a clearly defined interface for interacting with a system external to

[89] https://apisyouwonthate.com/

the application using, or *consuming*, the API. For instance, if you were building a location-based application, you might used Google's aforementioned mapping API to display maps and markers. If you wanted to manage uploaded files in the cloud, Amazon's S3 API is a commonly used solution. These APIs provide well documented endpoints which you'll use to interact with the service.

More recently, APIs have taken on a new role due to the rise in popularity of single page applications (SPAs). It's become increasingly common to separate a web application into two code bases: *frontend* and *backend*. The backend exposes an API capable of interacting with the application's data in all of the usual ways (e.g. retrieving events, registering users, joining events, etc.). The frontend is entirely driven by JavaScript (typically by way of a JavaScript framework such as Vue.js or React), and uses AJAX to communicate with the API. Further, because the frontend and backend are decoupled, the opportunity arises to create additional frontends such as a native iOS or Android application which can also talk to the backend API.

Web-based APIs not surprisingly rely on HTTP-based requests and responses, with data passed back and forth using a combination of URIs, headers, and messages formatted using JSON or XML. Once the API specification has been defined, the API maintainer will often implement those specifications within a library third-parties will then use to interact with the API. For instance, Amazon exposes most (if not all) of its services through the AWS SDK for PHP[90].

If you're in need of an API fast, and don't have the time or inclination to build one check out DreamFactory[91]. I recently had the pleasure of working with the company, and can confirm they've built an incredible product which allows you to expose a database through a REST API with no programming involved!

Introducing Laravel's API Infrastructure

All new Laravel applications include infrastructure intended to facilitate API integration. It can however be confusing to newcomers so let's take a moment to review the key components. For starters, API-specific routes are defined within the `routes/api.php` file. Open this file and you'll find a single predefined route:

```
1 Route::middleware('auth:api')->get('/user', function (Request $request) {
2     return $request->user();
3 });
```

All routes defined within `routes/api.php` are nested within a route group which applies a URI prefix, middleware, and namespace. These settings are applied within the application's route service provider, located in `app/Providers/RouteServiceProvider.php`:

[90] https://aws.amazon.com/sdk-for-php/
[91] https://www.dreamfactory.com

```
protected function mapApiRoutes()
{
    Route::prefix('api')
         ->middleware('api')
         ->namespace($this->namespace)
         ->group(base_path('routes/api.php'));
}
```

The `prefix` method defines a URI prefix of `api`, meaning all API-related route URIs will be prefixed with `/api/`, such as `http://hackerpair.com/api/events`. You're free to change this prefix to anything you please, or remove it altogether.

The `middleware` method identifies any middleware which will be applied to the API routes. The default is `api`. The `api` middleware is defined in `app/Http/Kernel.php` within the `$middlewareGroups` array:

```
'api' => [
    'throttle:60,1',
    'bindings',
],
```

This middleware definition performs two tasks. First, it restricts the number of API requests a user can make within a specified period of time. The default is 60 requests in 1 minute. Therefore if you wanted to change the default limit to 100 requests in 5 minutes, you would change the `throttle` setting to `throttle:100,5`. Second, the `bindings` attribute enables route model binding for API routes.

The namespace is defined by the `namespace` method. By default this method accepts `$this->namespace` as the input parameter, which is set to `App\Http\Controllers`. This is in my opinion is an inconvenient default because it means web and API controllers would reside in the same directory. One easy fix is to modify the `namespace` input parameter to look like this:

```
->namespace($this->namespace . '\API')
```

This would cause Laravel to reference the `App\Http\Controllers\API` directory when mapping controllers to endpoints. Within `routes/api.php` you could then define your routes just as you do in `routes/web.php`:

```
Route::resource('events', 'EventsController');
```

If you'd rather not modify the route service provider, you can prefix `API\` to your controller names to achieve the same outcome:

```
Route::resource('/events', 'API\EventsController');
```

Finally, the `group` method applies the route settings to all routes defined in the `routes/api.php` file.

Returning to the default route defined within `routes/api.php`, which for convenience I'll reprint below, I'd like to clarify on another matter of great confusion:

```
Route::middleware('auth:api')->get('/user', function (Request $request) {
    return $request->user();
});
```

Based on what you just learned about the API route prefix, middleware, and group, you might be inclined to test this default route by logging into your project account and navigating to `http://hackerpair.test/api/user`. However this will fail, because Laravel doesn't know how to retrieve `$request->user()` since API requests are *stateless*. The user identity is retrieved via the `auth:api` middleware which as you can see is enabled with this route. When the `auth:api` middleware is referenced, Laravel will look to the *authentication guard* defined in `config/auth.php`. Open `config/auth.php` and scroll down to this section:

```
'guards' => [
        ...
    'api' => [
        'driver' => 'token',
        'provider' => 'users',
    ],
],
```

The API guard driver default is `token`, which is defined in `laravel/framework/src/Illuminate/Auth/TokenGuard.php`. Long story short, this guard is by default going to look at the `users` table for a field named `api_token`. This field will contain each user's unique API token, which is typically a long randomly generated string. Therefore to actually navigate to the example API route you'll first need to add this field to your `users` table:

```
$ php artisan make:migration add_api_token_field_to_users_table --table=users
```

Open the newly created migration and modify the `up` and `down` methods to look like this:

```
public function up()
{
    Schema::table('users', function (Blueprint $table) {
        $table->string('api_token', 60)->unique();
    });
}

public function down()
{
    Schema::table('users', function (Blueprint $table) {
        $table->dropColumn('api_token');
    });
}
```

Next, run the migration to add the `api_token` column to your project's `users` table:

```
$ php artisan migrate
Migrating: 2018_02_14_173748_add_api_token_field_to_users_table
Migrated:  2018_02_14_173748_add_api_token_field_to_users_table
```

With that done open your database and add a dummy value to your test user's new `api_token` field (e.g. `1234`). Then open the browser and navigate to:

```
http://hackerpair.test/api/user?api_token=1234
```

You should be presented with a JSON object similar to the following (formatted for readability):

```
{
        "id":1,
        "provider":null,
        "provider_id":null,
        "first_name":"Jason",
        "last_name":"Gilmore",
        "email":"wj@wjgilmore.com",
        "city":"",
        "state_id":null,
        "zip":"43016",
        "lat":null,"lng":null,
        "timezone":"America\/New_York",
        "title":"Laravel Developer",
        "handle_github":null,
```

```
        "handle_twitter":null,
        "bio":null,
        "created_at":"2018-02-14 17:46:19",
        "updated_at":"2018-02-14 17:46:20",
        "last_login_at":null,
        "is_admin":0,
        "api_token":"1234"
}
```

Notice the `password` and `remember_token` fields are not present in this output. This is because they are shielded from view thanks to the `User` model's `$hidden` property:

```
protected $hidden = [
    'password', 'remember_token',
];
```

Feel free to add any other fields to this array which should not be included along with a model's array or JSON representations (`provider`, `provider_id`, and `is_admin` are all candidates in this case since they are relevant to application internals).

Creating an API Endpoint

Now that you're acquainted with Laravel's API-specific infrastructure, let's create an endpoint which allows users to retrieve a list of upcoming HackerPair events. Begin by creating a new controller named `EventsController` (if you were following along with the examples in chapter 12, then this controller and its parent directory has already been created):

```
$ php artisan make:controller API/EventsController --resource
```

Notice I nested the controller inside a directory named `API`. Laravel will automatically create this directory for you if it doesn't already exist. Open this controller in your IDE and modify the `index` method to look like this:

```
public function index()
{
        return Event::with(['category', 'organizer', 'state'])
        ->orderBy('start_date', 'desc')
        ->take(5)->get();
}
```

Next, we'll need to add a new route to the application. However, you won't add this route to `routes/web.php`. Instead, you'll add it to the `routes/api.php` file:

```
Route::resource('events', 'EventsController');
```

With the controller and route in place, navigate to the new endpoint (e.g. `http://hackerpair.test/api/events?api_token=1234`) to view a list of five events. You'll receive a large amount of JSON data such as the formatted snippet presented here:

```
[
{
"id":2,
"user_id":11,
"category_id":14,
"name":"Dolores Inventore Et Et Deleniti.",
"slug":"dolores-inventore-et-et-deleniti",
..
"description":"the description.",
"deleted_at":null,
"category":{
        "id":14,
        "name":"R",
        "slug":"r",
        "created_at":"2018-02-15 17:53:15",
        "updated_at":"2018-02-15 17:53:15"
},
"organizer":{
        "id":11,
        "provider":null,
        "provider_id":null,
        "first_name":"Aliyah",
        "last_name":"Lemke",
        "email":"mona.casper@pagac.com",
        ...
        "updated_at":"2018-02-15 17:53:07",
        "last_login_at":null,"is_admin":0,
        "api_token":"5a860f53da7bf"
},
"state":{
        "id":29,"name":"Mississippi",
        "abbreviation":"MS",
        "created_at":"2018-02-15 17:52:56",
        "updated_at":"2018-02-15 17:52:56"
}
},
...
```

Return to the `Events` controller's `index` query and you'll see that in addition to each event, we're retrieving the related category, organizer, and state records. This is convenient for the client, however it isn't without peril. In particular, note the organizer's e-mail address and API token have been exposed! You can resolve this by either limiting the columns selected when using `with`, or block the columns by default by adding them to the model's `$hidden` property.

Of course, you're not required to respond with all of a record's relations. You might instead only want to return each event's name, start date, city, and state:

```
$events = Event::join('states', 'events.state_id', '=', 'states.id')
            ->select(
                    'events.name',
                    'start_date',
                    'city',
                    'states.name as state')
            ->orderBy('start_date', 'desc')
            ->take(5)->get();
```

This returns a much more manageable JSON set:

```
[
{
        "name":"Dolores Inventore Et Et Deleniti.",
        "start_date":"2018-03-22 00:00:00",
        "city":"Hauckport",
        "state":"Mississippi"
},
...
]
```

Try applying what you've learned here to creating an endpoint which returns just a single event. Hint: you'll update the `API/Events` controller's `show` method just as you did in earlier chapters.

Creating New Events via the API

In the last section we created an endpoint for retrieving events. What if you wanted to create new events via the API? The general process is effectively identical to that employed when we created new events via a form in earlier chapters, with two distinct differences:

- Because a form is not used, a CSRF token will not be generated.
- Manually testing the endpoint is slightly more difficult, again because there is no form.

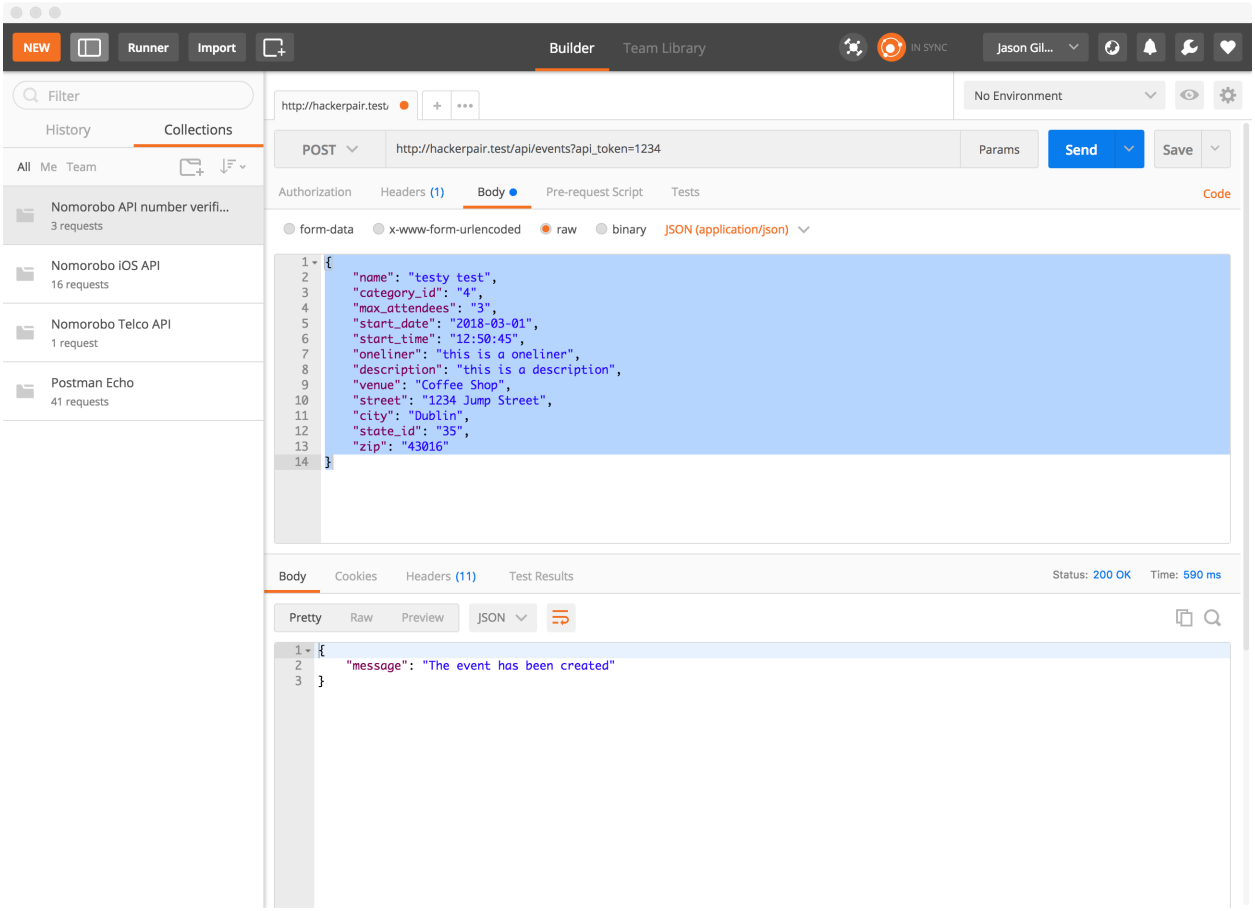
Let's begin by adding the creation logic to the API `Events` controller's `store` method:

```
public function store(Request $request)
{

    $user = User::where('api_token', $request->get('api_token'))->first();

    $event = Event::create(
        $request->input()
    );

    if ($request->has('published'))
    {
        $event->published = 1;
        $event->save();
    }

    $user->hostedEvents()->save($event);

    return response()->json([
        'message' => 'The event has been created',
    ], 200);

}
```

There's nothing particularly interesting here, other than we determine the event host's identity by querying the database for the provided API token, and we return a JSON-formatted response (logically you'll want to add additional error-checking and return a different status code in the event the record is not saved).

The event creation logic is in place, but how is the event information submitted to the endpoint? The client will send a POST request to the endpoint, passing the API token along as a URL parameter and the event information along in JSON format. When testing the endpoint you can use an API testing tool such as Postman[92]. The below screenshot depicts Postman being used to POST a new event to the HackerPair database.

[92] https://www.getpostman.com/



Sending a POST request using Postman

However, Laravel will by default expect a CSRF token to accompany the POST request. But no CSRF token exists, so you'll want to disable this feature when the API is used. Open the `app/Http/Middleware/VerifyCsrfToken` middleware and modify the `$except` property to include the `api` URI. This will disable the CSRF token requirement for all URI's prefixed with `api`:

```
protected $except = [
    'api/*'
];
```

With this change in place, try submitting a new event via a client such as Postman, and then check out your database to confirm the new record has been created!

Introducing Laravel Passport

The API token guard is fine for simple use cases in which a third-party must authenticate before interacting with your API. However there's a second solution which opens up a whole new world of

possibilities, not only for third parties but also should you desire at some point to begin converting your traditional web application into a SPA. This solution is known as Laravel Passport[93]. It too can generate unique tokens which users can then supply when communicating with the API. But Passport is so much more than that, acting as an OAuth 2 server for your application. I'll leave it to you to learn more about OAuth 2 specifics, and instead just enumerate a few of the features an OAuth 2 server has to offer:

- Third parties can build their own clients using your API, and point interested users back to your application for the authorization step. The users can login *on your server*, and be redirected back to the client application at which point the client application will be authorized to interact with your application on the user's behalf.
- Your users can login to your application using their account username and password, and receive an *access token* in return. This ensures a seamless login process without additional steps otherwise involved when a user logs in using a third-party client such as described in the above bullet point.
- Should you wish to separate your application frontend and backend as described at the beginning of this chapter, Passport can automate the generation and transfer of a *JSON Web Token* between the client and server.

Although in future revisions of this chapter I plan on expanding greatly upon this topic of Laravel Passport, for the moment I'm going to focus on one particular feature known as the *personal access token*. But first, let's install and configure Passport.

Installing Laravel Passport

Laravel Passport is available via Composer, so go ahead and add it to your application by running the following command:

```
1 $ composer require laravel/passport
```

In addition to installing and registering the package, several new database migrations will be available. The tables created by these migrations are responsible for managing the different types of tokens supported by Passport:

[93] https://laravel.com/docs/master/passport

```
$ php artisan migrate:status
+------+-----------------------------------------------------------------+
| Ran? | Migration                                                       |
+------+-----------------------------------------------------------------+
| Y    | 2014_10_12_000000_create_users_table                            |
| Y    | 2014_10_12_100000_create_password_resets_table                  |
| N    | 2016_06_01_000001_create_oauth_auth_codes_table                 |
| N    | 2016_06_01_000002_create_oauth_access_tokens_table              |
| N    | 2016_06_01_000003_create_oauth_refresh_tokens_table             |
| N    | 2016_06_01_000004_create_oauth_clients_table                    |
| N    | 2016_06_01_000005_create_oauth_personal_access_clients_table    |
| Y    | 2017_12_05_020610_create_states_table                           |
| Y    | 2017_12_05_162139_create_categories_table                       |
| Y    | 2017_12_05_163213_add_state_field_to_users_table                |
| Y    | 2017_12_05_202426_create_zip_codes_table                        |
| Y    | 2017_12_05_202516_create_events_table                           |
| Y    | 2017_12_05_203002_add_soft_delete_to_events                     |
| Y    | 2017_12_06_141645_create_favorite_events_table                  |
| Y    | 2017_12_06_151744_create_tickets_table                          |
| Y    | 2018_01_07_004655_add_is_admin_to_user_table                    |
+------+-----------------------------------------------------------------+
```

Notice these migration names are back-dated. It's no big deal and won't affect your existing database. Go ahead and run the migrations to create these new tables:

```
$ php artisan migrate
```

Next, run the Artisan `passport:install` command. This command will generate encryption keys subsequently used to create secure tokens used by your users and other clients:

```
$ php artisan passport:install
Encryption keys generated successfully.
Personal access client created successfully.
```

Finally, we're going to need to make a series of changes to various application files, beginning with the `User` model. Open the model and add the `HasApiTokens` trait:

```
use Laravel\Passport\HasApiTokens;
...

class User extends Authenticatable
{
    use HasApiTokens, Notifiable;
    ...
```

Next, open the `app/Providers/AuthServiceProvider.php` file and update the `boot` method to execute the `Passport::routes` method:

```
<?php

namespace App\Providers;

use Laravel\Passport\Passport;
...

class AuthServiceProvider extends ServiceProvider
{

        ...

    public function boot()
    {
        $this->registerPolicies();

        Passport::routes();
    }
}
```

Finally, recall the `config/auth.php` file we examined earlier in the chapter? The API authentication guard was set to `token`. This needs to be updated to use the `passport` authentication guard:

```
'guards' => [
    'web' => [
        'driver' => 'session',
        'provider' => 'users',
    ],

    'api' => [
        'driver' => 'passport',
        'provider' => 'users',
    ],
],
```

With these changes in place, you're ready to begin generating personal access tokens!

Generating Personal Access Tokens

Imagine you announce availability of the Hackerpair API beta release, and invite users to try it out. They'll need an API token similar to that we used earlier in the chapter, however this time we're going to leave it to Passport to do the work. Imagine a web interface which generates new API keys on the user's request. To accommodate this request, all you have to do is execute Passport's `createToken` method, which is included with user objects thanks to the earlier addition of the `HasApiTokens` trait. I'll simulate accommodating this request in Tinker:

```
$ php artisan tinker
Psy Shell v0.8.17 (PHP 7.1.8 â€” cli) by Justin Hileman
>>> $user = User::find(1)
[!] Aliasing 'User' to 'App\User' for this Tinker session.
=> App\User {#948
     id: 1,
     provider: null,
     provider_id: null,
     first_name: "Michael",
     last_name: "Roberts",
         ...
     last_login_at: null,
     is_admin: 0,
   }
>>> $token = $user->createToken('Hackerpair')->accessToken
=> "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIs...Um1Py-KdjXfQ"
```

This token is very long; I've drastically truncated it in the interests of space. The user would then pass this token along within the `Authorization` header of each API request.

Creating a Simple PHP API Client

To demonstrate use of the newly generated token let's create a simple PHP client. Create a new project directory and inside it create a file named `composer.json`. Add the following contents to it:

```
{}
```

That's right, all you need in this file is a pair of curly brackets. Next, add the Guzzle HTTP library to this project:

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

Once added, if you have a look at `composer.json` you'll see it now looks like this:

```
{
    "require": {
        "guzzlehttp/guzzle": "^6.3"
    }
}
```

Next, create a file named `index.php` in this project directory, and add the following contents to it:

```
<?php

require "vendor/autoload.php";'

$accessToken = "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIs...Um1Py-KdjXfQ";

$client = new GuzzleHttp\Client;

$response = $client->request('GET', 'http://hackerpair.test/api/user', [
    'headers' => [
        'Accept' => 'application/json',
        'Authorization' => 'Bearer '.$accessToken,
    ],
]);

echo $response->getBody();
```

Once again, I've truncated the personal token string dramatically for reasons of space. At any rate, this token is passed into the Guzzle request header, and the URL is set to `http://hackerpair.test/api/user` which was introduced earlier in this chapter. Save the changes, and then run the following command from inside this directory:

```
$ php index.php
```

You should receive a response similar to the following (formatted for readability):

```
{
        "id":1,"provider":null,
        "provider_id":null,
        "first_name":"Michael",
        "last_name":"Roberts",
        "email":"jdoyle@gmail.com",
        "city":"Cassandrechester",
        "state_id":51,"zip":"76481",
        "lat":"32.99042000",
        "lng":"-98.74549000",
        "timezone":"America\/Adak",
        "title":"PHP Developer",
        "handle_github":null,
        "handle_twitter":null,
        "bio":"this is the bio",
        "created_at":"2018-02-15 17:53:07",
        "updated_at":"2018-02-15 17:53:07",
        "last_login_at":null,
        "is_admin":0
}
```

Congratulations, you've just implemented a Passport-driven Laravel API and corresponding client! Try extending this simple PHP client's capabilities by parsing the JSON.

Conclusion

Laravel-driven APIs really do open up a whole new world of possibilities in terms of what you can do with your applications. Laravel Passports makes this process even more fun by removing a lot of the dreary coding you'd otherwise have to write and test. If you wind up building a new API using what you've learned here, please let me know about it! E-mail me at wj@wjgilmore.com.