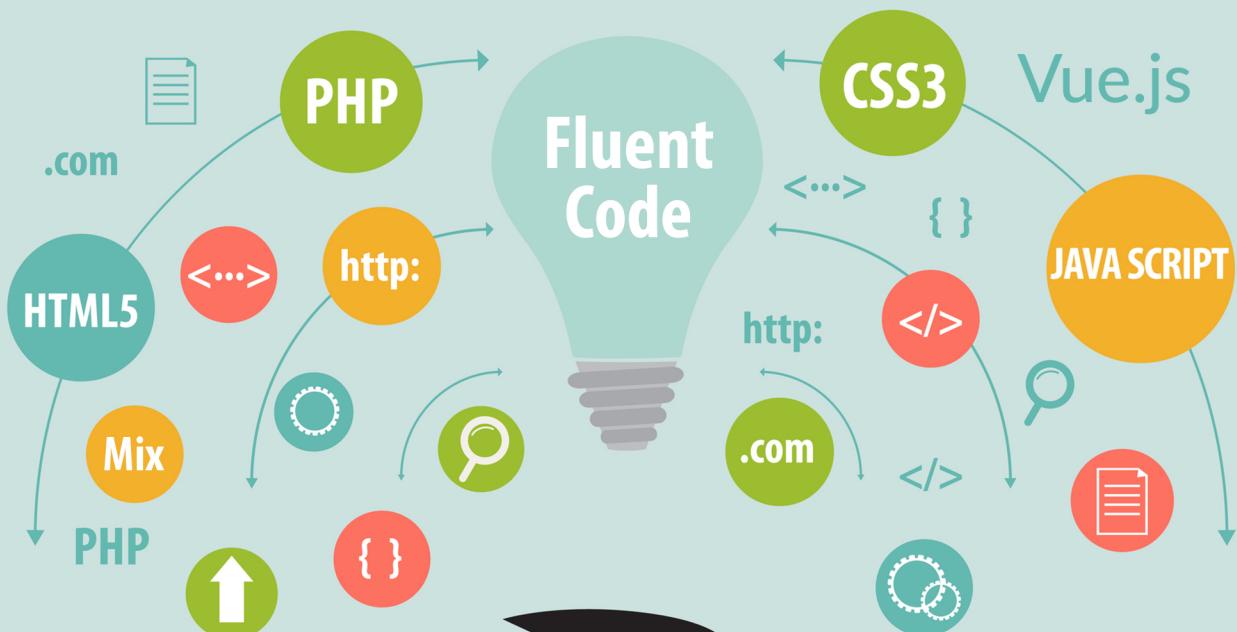


LARAVEL 5.4

FOR BEGINNERS



A step by step
guide to learning
Laravel

BILL KECK

Laravel 5.4 For Beginners

Bill Keck

This book is for sale at <http://leanpub.com/laravel-5-4-for-beginners>

This version was published on 2017-03-04



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2017 Bill Keck

“We need to illuminate PHP development, so let us turn on the lights.” –Taylor Otwell

Contents

Chapter 1: Introduction	1
Introduction	1
Features	1
What Makes The Laravel Framework Special?	3
Upsides	3
Downsides	4
Why I'm Writing This Book	4
Artisan	5
MySql	5
Documentation	5
Minimum PHP Skills	6
W3 Schools	6
Laracasts	7
Minimum HTML and CSS skills	7
Minimum Javascript Skills	7
Minimum Node.js Experience	8
Errata	8
Contact Bill Keck	9
Sample App on Github	9
Summary	10
Chapter 2: The Development Environment	11
Setting up the Dev Environment	11
MAC or Windows?	11
MAMP	11
IDE	12
Composer	13
Minimum Version of PHP	13
Node.js	14
Homebrew	14
Git	14
Command Console	14
Summary	15
Chapter 3: Installation of Laravel	16
Composer install of Laravel	16

CONTENTS

Create Project in IDE	17
Setup Local Host file:	21
Vhost Entry	21
httpd.conf	22
MOD Rewrite	23
Restart Apache	23
Summary	24
Chapter 4: Let's Get Started With Laravel	25
Set Up The Repository	25
Initial Commit	25
Diving Into Workflow	26
Setup the DB	27
.env	29
Application Structure	31
Basic Stitching	31
Routes	32
The Style Problem	37
Unit Testing	38
Creating a Route	39
Creating a Controller	41
Artisan	41
RESTful pattern	47
Namespaces and Use Statements	48
Index Method	49
Views	50
Blade	52
Creating A Master Page	54
CDN	59
View Partials	64
Full Code	67
Summary	68
Chapter 5: User Registration And Login	69
Controllers	69
User Model	70
Migrations	71
Make Auth	77
RegisterController	84
Traits	85
Gravatar	111
Facades	113
Pages Controller	114
New Nav	116
Auth Methods	123
Auth Views	136

CONTENTS

passwords view folder	137
email.blade.php view	137
reset.blade.php	145
Auth View Folder	151
register.blade.php	151
login.blade.php	156
Change redirectTo Properties	160
Summary	161
Chapter 6: Working with the RESTful Pattern	162
Sweet Alert	162
Usage	165
Flash Messages	166
Model	167
Model Factory	172
Route Resource	177
RESTful Controller	178
Errors List	180
Errors Folder	180
Store Method	187
Die and Dump	188
Index Method	192
Pagination	196
Create Button	198
Slugs	198
Changing the Store Method	200
Create the Slug	200
Changing the Create Method	200
Add Auth Use Statement	201
Change \$fillable Property on Widget Model	201
Basic Relationships	202
Add Middleware to WidgetController	205
Change Routes to Widget	207
Modify index view	208
New Factory method	209
Show Method	210
Accessors and Mutators	217
Edit Method	223
Update Method	227
Destroy Method	229
Automatic Route Model Binding	229
Error Handling	233
404.blade.php	233
Exceptions	235
Summary	244

CONTENTS

Chapter 7: Access Control	245
OwnsRecord Trait	245
Modify User Table Migration	259
User \$fillable	262
Admin Middleware	267
AllowIfAdmin	268
isAdmin Method	274
Admin Index	276
NoActiveAccountException	276
Modifying the LoginController	281
LoginController	284
Update Users Table	289
Registration Form	289
register.blade.php	290
Users Migration	296
Update User \$fillable	298
RegisterController	299
Create Method	300
Terms Of Service	301
Privacy	303
Summary	303
Chapter 8: Socialite - One Click Facebook Login	304
Facebook	304
Social Routes	305
session.php	307
Clear Cookies	307
InvalidStateException	307
Tip for nginx users.	308
Set up Facebook App	309
Step 1	309
Step 2	309
Step 3	309
Step 4	309
Step 5	309
Step 6	310
Step 7	311
Step 8	312
Step 9	313
Step 10	315
Step 11	317
Integrating Socialite	324
Exceptions	325
A Big Heads Up	326
AuthController	326

CONTENTS

Tip For Staying Current	327
ManagesSocial.php	337
Putting It All Together	351
Database Transactions	368
Summary	377
Chapter 9: Profile, Settings and Admin Dash	379
Profile	379
determine-profile-route and show-profile Routes	403
Add Profile to Nav	409
Profile Views	410
Show View - Profile	410
Index View - Profile	414
Edit View - Profile	416
Users For Admin	420
UserController.php	421
UserRequest	425
User Model changes	429
HasModelTrait	433
Index View	435
Show View	438
Edit View	442
Navigation to Users & Profiles	446
Settings	451
Settings Routes	452
SettingsController	452
Create Settings View Folder	455
Edit View For Settings	456
Add Settings To Nav	461
Admin Page	462
Add Facebook sign in buttons to Login and Register Views	466
View Composers	467
View Share	471
Summary	472
Chapter 10: Working With Images	473
Create imgs, marketing-images, and thumbnails folder.	474
Create Image Request	495
Show Method	510
Add Display Methods to MarketingImage Model	511
ShowsImages trait.	512
show view	513
Edit view	518
Edit Method	524
Edit Image Request	525
Update method	527

CONTENTS

Destroy Method	532
Index method	533
index view	534
Add Marketing Images To Nav	537
Carousel	538
Pages Index View	538
Update slider.blade.php	545
image_weight	551
Modify MarketingImageController	553
CreateImageRequest and EditImageRequest	554
Marketing Image Create and edit Views	554
MarketingImage Show View	556
PagesController Index Method	557
grid.blade.php	558
MarketingImageController Index Method	558
marketing-image index.blade.php	559
Summary	561
 Chapter 11 Introducing Mix and Vue.js	562
Node	562
NPM	563
Running NPM Install	563
Compiling Multiple Assets	572
Versioning	575
Vue.js	578
Vue Basics	586
Summary	599
 Chapter 12: Data Grids with Vue.js	600
Datagrid	600
Api Route	600
ApiController	600
Implementing A Contract	617
main.scss	619
WidgetGrid.vue	634
MarketingImage Data Grid	656
API route	656
marketingImageData method on Api Controller	656
MarketingImageQuery.php	657
MarketingImageGrid.vue	659
npm run dev	674
MarketingImageController.php	674
marketing-image/index.blade.php	674
Summary	681
 Chapter 13: Events, Mail, and Architecture	682

CONTENTS

Mail	682
Email Confirmation On Registration	682
config/mail.php	686
Passing Data to the Email	688
Markdown Email	690
Custom Themes	695
Events	697
Registering the Event	697
Naming Events	699
event:generate	700
ShouldQueue	706
Application Structure	708
The Service Container	708
Automatic Injection	718
Method Injection	718
Constructor Injection	720
Service Providers	720
Aliases	732
Facades In Realtime	733
Summary	735
Chapter 14 Chat with Laravel Echo, Vue, and Pusher	736
Vue.js Nested Components	736
Chat	751
Routes	751
Chat Controller	752
Message Migration	754
Message Model	755
Eager Loading	756
Query Scopes	757
Messages Method On User Model	758
Messages Factory	759
Chat Front End	763
Chat Index	763
components.js	765
app.js	765
ChatList.vue	773
ChatMessage.vue	776
Installing Moment.js	779
ChatCreate.vue	782
Pusher	788
Laravel Echo	791
Console Command	803
Summary	814
Chapter 15: Custom Validators and Vue.js Dependent Dropdown	815

CONTENTS

Setting Up The Lesson	815
Category Model	816
Category Migration	816
Category Controller	818
Category Views	821
category/create.blade.php	821
category/edit.blade.php	823
category/index.blade.php	825
category/show.blade.php	826
CategoryGrid	830
components.js	840
Add categoryData method to ApiController.php	840
CategoryQuery.php	841
Category Factory	842
Subcategory	843
Subcategory Model	843
Subcategory Migration	844
Subcategory Controller	845
Subcategory Views	849
subcategory/create.blade.php	849
subcategory/edit.blade.php	852
subcategory/index.blade.php	855
subcategory/show.blade.php	856
SubcategoryGrid.vue	859
components.js	870
ApiController	870
SubcategoryQuery.php	871
Seed data	872
Route	874
Lesson Model	875
Lesson.php	875
Lesson Migration	876
LessonController	877
LessonController.php	878
Lesson Views	882
lesson/create	882
lesson/edit.blade.php	884
lesson/index.blade.php	886
lesson/show.blade.php	887
LessonGrid.vue	891
components.js	901
ApiController	901
Api Route for Lesson Model	902
LessonQuery	902
Custom Validation	908

CONTENTS

Lesson Create Request	908
Validator Service Provider	913
Multiple Custom Validation Rules	915
Left Join	918
LessonQuery.php Revised	918
LessonGrid.vue Revised	920
Lesson Show Revised	921
Dependent Dropdown	922
LessonCreateCategory.vue	923
lesson/edit.blade.php	931
LessonEditCategory.vue	935
Summary	940

Chapter 1: Introduction

Introduction

Welcome to Laravel 5.4 for Beginners. If you are reading this, you've no doubt heard about the Laravel 5.4 PHP framework, the leading PHP framework in production today. It's a wonderful framework developed by Taylor Otwell, which he started back in June 2011. Taylor has evolved the framework quickly, and whereas in 2011, it started as a skinny new kid on the block, 6 years later, it is now an enterprise-level powerhouse.

Taylor somehow synthesizes complex design patterns with easy-to-follow syntax and concepts that make coding in Laravel a pleasurable experience, at least once you are at the point where you understand it and get it working. I'll get you to that point. Taylor's intuitive syntax will help you on that journey too and in the end you will love the framework because of it.

My goal with this book is to help you get started, so you can begin successfully developing in Laravel. If you are a beginner now, I want to help you get to a more intermediate level.

I'm mindful of the fact that you aren't just learning programming for fun, that it is a commercial skill, and that the quicker you learn it, the more financially successful you will be.

You may have noticed my other book on Laravel 5.3. Before that, the book was written for Laravel 5.2, and before that, 5.1, so this is actually my fourth pass on a book on Laravel.

Rather than write a new book, I would prefer to update the old books, but that isn't practical because things are evolving quickly. Every time I work on it, the authentication traits change. Since we do a Socialite implementation for Facebook one click login and registration, this always requires a lot of rewriting.

Another example is 5.3 introduced a sample component for Vue.js out of the box, which led me to cover Elixir, the previous version of Laravel's front-end asset manager.

In 5.4, Elixir has been completely replaced by Mix. Mix is very similar, but there are differences in the files that are used.

With each new version of the book, I add bonus material specific to the new version and will be doing that for this book as well, going a little deeper with Vue.js.

Vue.js is a front end javascript framework that is seeing wide adoption, so it makes sense to cover more of it. Don't worry if you are not familiar with asset management, we will cover Mix and Vue.js in detail in later chapters.

Features

Laravel is a full-featured framework and you'll be amazed at what you can do with it. Some of the features you get out-of-the-box include:

- pre-defined schema for the user table

- user login and registration
- forgot password and login throttling functionality
- automatic code generation for migrations, models, controllers, views and other classes
- Eloquent ORM for working with databases
- Blade templating engine to make front end integration easy

If you don't understand something in that list, don't worry, we will be covering it in detail. Just know that it really is amazing all the features you get, and that doesn't even cover some of the packages we will be using that plug right into Laravel.

We will be using:

- Socialite for one click Facebook registration and login
- Mix and Vue.js
- Sweet Alert for pretty ui on alerts
- Gravatar Plugin
- Intervention Image Management Plugin

With all of that we are going to build a sample project that has:

- backend admin
- access control
- user profile and settings
- requires active status for login
- redirects on login depending on Admin or User
- image management

Basically, it's a list of commonly used attributes that many sites will have. The idea is that you can use the knowledge you gain here as a typical starting point for projects.

When you are done with this book, you will have gained a wide range of experience in working with Laravel that will help you in your future application development.

We're also going to cover some Vue.js because of its popularity within the Laravel community and the way it is now integrated into Laravel.

I think it's very important for me to provide you with code for a working data grid, and while it would be easy to simply include the jQuery Datatables library, like I did in some of the previous versions of the book, it's not something I can recommend anymore. So we will venture gently into Vue.js, learn the basics, then tackle a full-featured datagrid and a chart.

Javascript is not necessarily a strong point for me, but I'm competent enough to get you up and running. If you want to understand javascript and the Vue.js library better, you need to do that on your own, and you may want to pick up a copy of [The Majesty of Vue.js 2](#).

This is primarily a PHP book, perfect for beginning PHP programmers who are ready to move on to framework development. We will start with basics and move towards more intermediate programming as the book progresses.

Advanced PHP artisans will be able to zip through this book and become framework-literate quickly. This will not only save them time on projects, but also fully leverage the benefits of an open-source framework that has an entire community of ninjas behind it.

The main style of this book, however, is for beginners. There's a lot of granular detail to help people who have some experience with PHP but don't have as much experience working with a framework yet. It's ok, we've all have to start somewhere.

We take it step by step and focus on clear explanations with practical results. You end up with a working CRUD application.

What Makes The Laravel Framework Special?

Taylor Otwell brought something special to Laravel and he does a much better job of explaining the architecture than I ever could. If you get a chance, check out the following video for a sample of that:

[Taylor's Talk](#)

I had the feeling watching Taylor's talk that half the audience didn't have the slightest clue about what he was talking about. Don't worry if you feel the same way. All of this will make a lot more sense to you once you start working with Laravel.

While it may be hard to wrap your head around Taylor's technical descriptions if you are a beginner, the practical effect of the framework is phenomenal. You can see many pieces of a puzzle that came together in just the right way, and when you are using it, it somehow makes your job easier, which is a huge benefit of the framework.

I can't tell you how many times I've said, "It can't be that easy..." And yet it is.

As you are probably well aware, programming in general has some very formal and complicated aspects to it, things like design patterns and concepts like dependency injection, which can be confusing to a beginner. The shame of it is that Laravel is living proof of the power of design patterns played out at scale, with a simple sensibility in its syntax.

Now I just said a few words that sound nice, but what do they mean? Well, if you have ever struggled with parts of programming, let's say for example, writing queries, or something else like form validation, then your life is about to become more enjoyable.

Laravel makes things that can otherwise be incredibly complicated simple to the point of being trivial. That's still a little generalized, but let's leave it there for now, I will expose the meaning in all of this through implementation. Just know that you will love it.

Upsides

At this point in the game, most people know the benefits of using a PHP framework as opposed to not using one, but let's just take a quick look to be sure.

Here are some of the obvious benefits:

- Uses standard ways of doing things, so reduces or eliminates spaghetti code.
- Reduces time spent on plumbing tasks such as form validation and security.
- Makes it easier to work as a team by enforcing standards.
- Makes it easier to maintain code by utilizing a common architecture and methods.
- You get the benefit of an active community of developers who maintain the framework and support common tasks and new features.

Downsides

With Laravel, honestly, there isn't much to put in this section. Frameworks consume server overhead because of the way PHP runs and the extra bandwidth that an ORM consumes.

Much of this however is a moot point because PHP 7 eliminates most of that and as long as you use your ORM wisely, you do not need to worry about it.

Why I'm Writing This Book

Let me pause for a moment to talk about why I write programming books. I had a wonderful experience in writing my first book in that it helped programmers from all over the world, who wanted to learn Laravel.

Their positive comments, feedback and reviews inspired me and drove me to work as hard as I could to make the book as good as I could possibly make it.

The readers rewarded my work with positive reviews and recommendations. I really appreciated it. It made me feel like the long hours of writing were worth it.

Since then, I've talked to a lot of programmers about programming. The most common experience I hear is that learning programming is difficult and painful, frustrating on a level that often defies sanity.

I can relate to this because programming doesn't come easily to me. I wish it did. I spent a long time lagging behind in my development as a programmer because I couldn't find clarity in the tutorials and documentation that often seemed intentionally obscure.

I found plenty of tutorials that will show you how something works without explaining why it works. But the why is important. It provides context. And until I got to a certain level with programming, people seemed unwilling to help me understand it, almost rude in their responses to questions.

On the other hand, there were some programmers that really were willing to go the extra step. Ever since then, I've made it my mission to try to be as clear as possible with my tutorials and books, to try to explain the "why" behind what we are doing.

So let me sum up by saying I'm committed to helping you learn Laravel, committed to helping you learn how to build worthwhile features for your applications, because I know exactly how you feel when you get frustrated, if something you are trying to learn isn't clear. I will do my best to make things clear and be as helpful as I can to help you meet your programming goals.

Artisan

Laravel has this wonderful command line tool name Artisan. You will find yourself typing the following on the command line:

```
php artisan make:
```

I will explain how we use this tool in detail, but notice the choice of words in the command. Imagine you are the artisan in command of the code. You are the php Artisan. It's a nice feeling. Eventually, you will experience this for yourself.

I just wanted to point this out as an example of Laravel's superior syntax. It reinforces your relationship to the work. It's baked into everything you do and promotes a pleasurable coding experience. These little syntactic hooks work wonders...

MySql

Throughout this book we use Mysql as the database, which, in addition to being free, is capable of powering enterprise data for web applications.

Because of the structure of the database, with its indexes and primary keys, a database can serve data very efficiently. In the simplest terms, this means it is very fast. It's also very deep. It can hold millions of records, which can be retrieved, if structured properly, in milliseconds.

Another key aspect of the database is that it allows us to structure the data in such a way as to connect things like the user's address and their username as if they were one record, but hold them in separate tables as separate records. The more you can break down the data structure into discrete components like that, the more powerful it is. This is called normalization of data.

The problem is that the more refined the database is, the more effectively normalized it is, the more complex it is to deal with in PHP. You end up having to connect a lot of PHP models together to represent the data correctly.

Now this might be getting too granular for an introduction, so we won't take this much further for now, but the point is to understand the nature of the problem that the framework helps to solve. The easier it is for you to connect the models via the framework, the more power you derive from your database. Laravel is awesome at this task as we will see later in the book.

Documentation

Taylor Otwell personally wrote the [documentation for Laravel](#) and he did an excellent job. His feel for programming syntax is matched by his feel for English syntax. The docs are well-organized and easy to understand.

It's important for you to be used to referencing the docs because you will have to do so often, unless you have a photographic memory. I will show you how great the docs are later in the book when we are in workflow.

I don't know if you will do it, but I highly recommend you read the documentation start to finish, it will help you be familiar with where everything is. Don't worry if you don't understand it all, we will explore things in detail in this book with specific examples.

Minimum PHP Skills

Franz Kafka once said, "The Messiah will only come when he is no longer needed..."

It's another way of saying you only get the answer when you no longer need it. Learning a PHP framework is simple as long as you are a skilled PHP programmer. Quite often PHP is a second or third language for a programmer who is already proficient in an object oriented language like C or Java, so learning PHP is just a matter of adjusting syntax and they pick it up quickly. That's great for ninjas, but what happens if you are just learning your first programming language?

I've already mentioned how difficult learning can be, and as someone who learned PHP as a first programming language, I can tell you from direct experience that moving from beginner to advanced is hard because there is not a lot of support for the middle ground.

If you search online, you either find incredibly complex examples involving multiple interfaces with nested objects or examples that are so rudimentary that, while they are easy to understand, they do nothing to advance your abilities.

Working with Laravel will help you enormously along this path, but you need some understanding of object-oriented PHP before diving in. You can get this from a variety of sources online.

I got my first taste of PHP from thenewboston.org, which has 200 videos on PHP. Great for an introduction, but not much more, since it's kind of dated at this point. A much better choice would be [The PHP Practitioner Series](#).

I followed up my initial exposure to PHP with a quick read of Richard Reese's book on Java, which helped me understand object-oriented programming better, since everything in Java involves a class. Also, when I looked at PHP again, it seemed simpler. I also went through the basics at:

[W3 Schools](#)

W3 Schools

W3schools.com is a great learning resource. You can play with the code online at that site.

And then of course there is:

[PHP.net](#)

PHP.net is where we find all the docs for the language and sometimes very complicated examples. I learned a lot there and got lost a lot too, that's the way it goes. Try it, you'll see what I mean.

At any rate, to be able to work with Laravel, you should understand the basics about objects, arrays, and control structures like foreach loops. You should know the components of a class, properties and methods, etc. Take a look at:

[OOP for Beginners](#)

You should be able to get through that tutorial very easily. If not, go back and study it before trying to tackle Laravel. Also, Laravel uses PHP 5.6.4 and above, which supports new array syntax and namespaces, both of which will be utilized extensively. I never use the old array syntax.

If you are light on programming experience, but full of enthusiasm, you should do well, as long as you are willing to do the work and are patient. At any point, if you don't understand something, you can stop and take the time to research it on Google or stackoverflow or PHP.net.

PHP is a well-documented and well-supported language, used by countless programmers who will try to help you. Laravel itself is well-documented and well-supported.

Also, I took a lot of care to label the subsections of this book, so you can easily find what you are looking for, if you need to refer back to it. Many times you will want to return to a section to reference something and I've done my best to make that as intuitive as possible.

Laracasts

I would be remiss if I didn't mention here how great [Laracasts.com](#) is for learning programming. I don't miss an episode.

I know that education dollars can be hard to come by, and Laracasts, which does have free videos, also has paid videos. For programming instruction, it's about as good as it gets, and it's one of those things I just feel I can't live without.

I don't have any personal relationship with Laracasts, but I give it my highest endorsement. My company does have a company license for Laracasts and all the programmers there watch it.

Minimum HTML and CSS skills

The purpose of building our template is to serve it on the web, so it's inevitable that we will use html and css. If they were handing out prizes for the world's worst front-end designer, I would probably be in the running to win.

Luckily, with Twitter Bootstrap, developing the front-end is a lot easier than it used to be, and because of that, we get a layout that is mobile responsive and very professional looking.

I don't go too deep in teaching html, so the more familiarity you have with it, the more you will understand it. That said, all the html code you need for our sample application will be provided.

Minimum Javascript Skills

I do my best to try to keep the use of javascript as minimal as possible. Obviously, since we are using Twitter Bootstrap, we have a dependency on jQuery. Laravel makes this integration easy. I provide full instructions on how to set that up, so you don't have to be an expert with javascript or jQuery to follow the book.

As I mentioned previously, we will work with Vue.js, which may be new to some of you. Fortunately, Laravel comes with an example Vue.js component to show you how it works.

I'm going to provide the complete javascript code for both a datagrid and a chart, some of which is a little complicated. I will step you through that code in detail, but it would be great if you augmented that with some tutorials on Vue.

As I already mentioned, there is a book named [The Majesty of Vue.js 2](#) that I really recommend if you want to read up on Vue.js. I found it a great way to get started, with simple examples to work with.

For the basics of Javascript, I wholeheartedly recommend [Code School](#). It's a subscription-based site, so it's not free, but it's worth the money. It covers javascript in depth. I've personally watched all of their javascript roadshow and best practices series and it really helped fill in the blanks for me.

Minimum Node.js Experience

Towards the end of the book, we use Node.js, so will need to have that installed in your dev environment. The reason that I put it at the end is that if you have trouble with the setup, you can still benefit from most of the book.

Installing Node.js is simpler these days and I will provide links for that at the appropriate chapter.

We don't do anything special with node, Laravel itself does all the heavy lifting, but you do need to have node installed in your dev environment. Laravel handles almost everything else from there with very little configuration on our part.

Errata

Although I have poured over every line of code in this book at least a hundred times and built the examples from scratch twice just to make sure I could follow the directions, mistakes are bound to happen, such is the nature of technical writing. I am actively updating errata as I go, so I do hope to be able to quickly correct any errata I am made aware of. You can help by emailing me if you find something, everyone will appreciate it.

Formatting Tip:

In certain cases, I had to format my code using two lines where one would be appropriate, in order to avoid line breaks from the word-wrapping in PDF and other formats. The word-wrapping in PDF causes special characters to appear, which break the code, so I had to avoid that the best I could. As a result, I'm not recommending you follow the code examples as an example of style. I would recommend following the PSR-2 Guide, available here:

[PSR-2 Coding Style Guide](#)

I will also be supplying Gists for each block of code that we write in the book, where the block of code exceeds 3 lines. These come directly from my IDE, so it is verified working code and the format is much better than the book because I don't have to dance around the wordwraps. If you don't know what a Gist is, don't worry, we will cover it in detail later.

Contact Bill Keck

LeanPub provides a contact link at the bottom of the book's landing page, so feel free to contact me there:

[Email Bill Keck](#)

You can also leave a comment for me at my Laravel Tips blog:

[Laravel Tips](#)

Please feel free to leave a comment.

Please note the purchase price of this book does not include technical support. That said, you can reach me at the contact email I provided above and I am actively following the Laravel Tips blog, and generally, I do respond to questions.

The fastest way to overcome errors is to Google it, most likely someone has come across the same problem. Also, please keep in mind that Laravel is continually being developed and new versions may not support the code offered in this book. This is not unusual for programming books.

I'm going to do my best to stay on top of that, but there can be times when Laravel has made a version change that I have not accounted for yet. Once I'm aware of an issue, I can typically fix it quickly, so please do your part and notify me if you notice a versioning problem.

I will mention it numerous times and in numerous places that updates to this book are available to you for free for the life of the book. Simply login to your leanpub.com account to get the latest version.

Update notices for major updates go out by email. Minor updates simply get published, and these will typically just cover a typo.

To see if you have the latest version, you can go to the leanpub.com landing page for the book and look at the last updated date, which will show you when my last commit was.

Also, beginners will face a high volume of error messages due to typos and missed code on their part. It's perfectly fine and part of the learning process. You will learn more from troubleshooting bugs than you will from simply copying and pasting code.

In most cases, you will find the answers to your problems if you are patient. Laravel forums are an excellent source of support and there are many great programmers that will help you. Always do your best to try to solve the problem first because it would be foolish to tie up a programmer's time with support requests over typos. Nevertheless, that is bound to happen. Just remember to be polite and considerate of others and you will do great.

Sample App on Github

I have the repository for the sample app live on Github here:

[Github Repository](#)

That is the repository for the full template. You might want to favorite it.

Something to keep in mind there. First, if you are working along in the book, use the book code or even better, use the gists I supply because that will be the code relevant to the chapter you are working on. The repository is the completed project, so there may be differences as you work along.

Summary

I know it can be a little intimidating at first, especially when you realize that Laravel is not just some trivial set of library files that you can master in a few days, but hang in there and be patient. We are going to tackle it one step at a time.

So let me conclude the introduction with the following words of encouragement. You can do this. Just stick with it and move at your own pace. And soon you will be amazed at how you are using Laravel to power your applications and you will be even more amazed at what you can create with it.

Chapter 2: The Development Environment

Setting up the Dev Environment

Let's just recognize that setting up your development environment is never the fun part of the project. Sometimes it's the most difficult part. No matter which type of computer you are on, environment setup can give you grief if you haven't done it before. Quite often it gives you grief even if you have done it before.

The only reason I'm mentioning this is to help you to stick with it. If you have problems with setup, just google the problems and the answers will be there. I provide general instructions in this book, but I don't cover setting up environment variables in Windows and I don't cover the installation of the programs we will be using other than Laravel itself. Here's why. There's no way I can keep up on all the installation instructions and versions of environments and you are better off getting the instructions directly from the sources themselves. I will point you in the right direction, but then you are on your own. Be patient and persistent, you will find what you need.

MAC or Windows?

The first question to address concerning your development environment might be are you using a Mac or Windows? The Laravel community leans heavily toward Mac. I'm using a Mac.

If someone were to ask me to recommend a platform, I would strongly suggest using a Mac. But not everyone can just decide to get a Mac and some people prefer a linux machine or Windows. As a courtesy to those programmers worldwide who may be using windows, I will try to make helpful suggestions.

You can follow along the book in a Windows environment, you just have to be mindful of the differences in setting up. Once setup is done, it doesn't matter, in terms of this book, which one you use.

Advanced programmers are also likely to develop directly on Linux. I don't provide instructions for any kind of linux platform, but if you are using Linux, you probably know your way around all of this anyway.

Regardless of platform, we need to be able to serve PHP, Apache(or equivalent), and MySql.

Because I currently use it personally, I'm recommending MAMP, which creates a virtual server with everything we need, except for Composer, which is a separate install.

MAMP

MAMP includes PHP, Mysql, Apache, and PhpMyadmin, so it's perfect to create a development environment for your projects. It's also free.

[Download MAMP](#)

You could go with Homestead, which is part of the Laravel ecosystem that creates a local server environment. It's built off of Vagrant, but it doesn't include Php MyAdmin, and I use that quite often, so I'm recommending MAMP. It's free to use and in terms of ease of download and installation, I think it's a happy medium.

More advanced users may prefer Vagrant. If you have a different php development environment that you are comfortable, that's fine. You just need to be able to run PHP, Apache(or equivalent), MySql and Composer and have access to the command line.

Laravel ships with a program named Tinker, which allows you to play with the code on the command line. Back when I tried it when I was using Windows, it didn't work well with Windows, but I didn't see any mention of that in the docs, so that left me frustrated.

Switching to Mac, I found that Tinker is useful, but I only use Tinker for database seeding in my workflow, so I don't cover it in depth this book, which means you don't have to worry about it if you are on Windows. You will be able to seed your data without Tinker if you are on Windows.

When I'm experimenting with code, I usually do it through a test controller, and that is something we will cover later in the book.

Another important thing to note if you are using Windows is that you need to set the environment variables so that Windows knows the path to PHP and to Composer. If you are using Windows, you will have to set that up. Google instructions for setting up MAMP and Composer on Windows.

Luckily, MAMP for Windows is fairly easy to install, and if you can get it recognizing PHP and Composer, you should be good to go.

You can use any alternative to MAMP that provides a development environment compatible with PHP and a web server. It's up to you to decide which one you want to work with.

There are a number of tools that I recommend that you use for development in Laravel, some cost money, some are free. These are recommendations only, not necessary to follow exactly, but my instructions will assume you are using them.

If you are advanced enough, you can use whatever you wish, no big deal. As long as you have working development environment, you are fine. If not, try to use these exact tools, it will be easier for you in the long run.

IDE

For my IDE, I use Php Storm. IDE stands for Integrated Development Environment, and helps you organize projects and code. Most developers use some form of IDE as opposed to just a text editor. I'm also recommending Eclipse or Netbeans for this project, however, because both are free whereas Php Storm is a paid IDE.

In most cases, installing an IDE will require a current version of the JAVA SDK. If required, your install should prompt you for that.

[Download Eclipse](#)

[Download Netbeans](#)

[Download PHP Storm](#)

While I don't use it personally, many professional developers use Sublime text editor as their IDE.

[Sublime](#)

I've heard great things about it, but I haven't used it yet. The choice is ultimately up to you.

I recommend taking the time to really learn how your IDE works. There is a fantastic series on Laracasts about PHP Storm:

[Laracast PHP Storm](#)

It's for an older version of the IDE, but you can get a lot of great ideas from the series about what you can do with your IDE.

One day I spent 5 hours adjusting the color scheme of my IDE, developing my own theme. At the time, I thought it was crazy, but in the long run, it really helped me learn the IDE and recognize elements of code more easily. I highly recommend that you dig in deep to your IDE.

I should note that PHP Storm does have some quirks when it comes to Laravel, it has trouble recognizing the source of some of the classes. That means it will show you a warning on some of the classes. It won't stop you from coding. There is a patch available, but it's not perfect, so I adjusted my warning notice to be less obvious in the IDE because of it. Other than that, I really love PHP Storm.

Composer

Laravel, like most modern PHP Frameworks, requires Composer for autoloading and package management. You can get the full installation instructions for Composer [here](#):

[Download Composer](#)

You need to have curl enabled in your PHP build to follow these instructions. See the next section for how to check that.

Minimum Version of PHP

Required PHP 5.6.4

If you want to check your configuration in php, after setting up MAMP and your IDE, setup a php page you can view from the browser with the following code:

```
<?php  
phpinfo();  
?>
```

That will output the info on your PHP build. As noted, you will need a minimum of php 5.6.4 to run Laravel, so make sure you are using at least that version. You can also see if curl is enabled.

You will also need to set an environment variable for PHP if you are using Windows. If necessary, Google for instructions on how to set that to your version of Windows.

Node.js

Required, install globally. But we will only be using Node towards the end of the book, so you wait until then. You will also need to install npm, node's package manager.

You can download Node from:

<https://nodejs.org/en/>

Installation video:

[Installing Node](#)

Homebrew

Optional.

Many Mac developers prefer to use Homebrew to install programs, so I am mentioning it here:

[Homebrew](#)

Git

Optional.

I also recommend using git, which provides version control. Version control is a handy way of saving your work so you can step backward easily if you need to. When you are dealing with a large number of files that are constantly being updated, this is a great help. Git also protects you in a team environment from someone overwriting your work because you can simply step back to a previous version.

[Download Git for Mac](#)

[Download Git for Windows](#)

If you are not familiar with Git, take the time to learn about it, you will be glad you did.

Command Console

Optional.

Lastly, I recommend iterm2 for Mac, which is a command line tool that is a little prettier than the standard prompt. This makes it easier on the eyes and just a little easier to work with.

[Download iterm](#)

Summary

I provided links and reference pages for installation, but for beginners, this may prove to be difficult. You can use the installation of the development environment as one of the tests to see if you are ready to tackle Laravel.

Just don't give up easily. If it doesn't go well, you can always get help from a more experienced programmer.

Also, and this is a tip for beginners, almost everything you will go through as a programmer has been gone through by other programmers before you and this is especially true for configuration errors. Don't be afraid to use Google and StackOverflow for help in troubleshooting setup.

Before you move onto installing Laravel, make sure you have your development environment operational. Once you've got everything up and running, spend a little time learning your way around the tools. It will make your efforts developing in Laravel go a lot smoother.

You should know how to use MAMP, how to stop and start apache, how to view the PHP MyAdmin page, and how to create a database. You should know how to set the username and password of the database, the user being root.

If you are using a different development environment, that's perfectly fine as long as you can serve apache, php, and mysql.

Ok, let's move onto installing Laravel.

Chapter 3: Installation of Laravel

Ok, so we have one last setup chapter to get through. If you have your development environment working, it should go smoothly because Laravel itself is incredibly easy to install.

We are assuming at this point that you have MAMP(or some environment equivalent) and composer installed at this point.

If you have decided to proceed using Windows, no problem, just be mindful of the differences. I will try to point them out when I can.

If you prefer to reference the Laravel Docs for installation, you can get them here:

[Laravel Installation Docs](#)

Composer install of Laravel

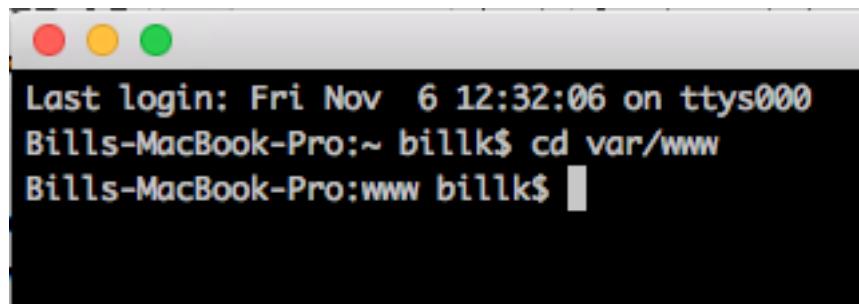
On your command line, change the directory to the root directory where you will keep all your projects. Where you put that is up to you.

For example, if you have a folder named www inside of your var folder, you would CD to var/www. You don't have to be in var/www, it can be a folder of your choosing, such as an htdocs folder. However, my examples will be for var/www.

The command on your command line is:

```
cd var/www
```

On a Mac, it looks like this:

A screenshot of a Mac OS X terminal window. The window has three colored title bar buttons (red, yellow, green). The terminal text area shows the following output:
Last login: Fri Nov 6 12:32:06 on ttys000
Bills-MacBook-Pro:~ billk\$ cd var/www
Bills-MacBook-Pro:www billk\$
The cursor is visible at the end of the third line.

The screenshot shows a Mac OS X terminal window with a dark background. The title bar is white with three colored buttons (red, yellow, green). The terminal text area contains the following text:
Last login: Fri Nov 6 12:32:06 on ttys000
Bills-MacBook-Pro:~ billk\$ cd var/www
Bills-MacBook-Pro:www billk\$
The cursor is located at the end of the third line.

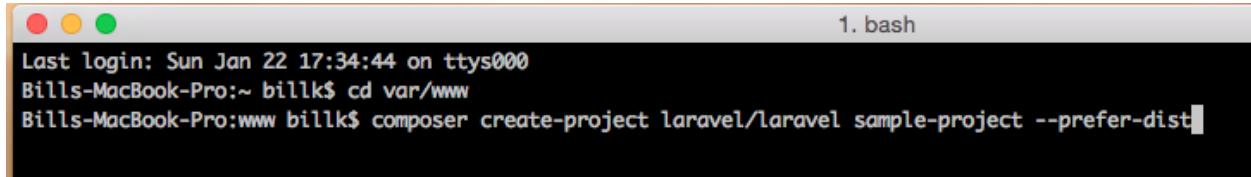
Note, if you are using windows, the command is with cd \, then cd \var\www or equivalent if you are using different folders.

Once you are in the correct folder, we use a simple composer command to install the project. This of course assumes you have composer successfully installed. If not, please install that first.

When you are ready, run the command like so:

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

Your command line should look like this:



```
Last login: Sun Jan 22 17:34:44 on ttys000
Bills-MacBook-Pro:~ billk$ cd var/www
Bills-MacBook-Pro:www billk$ composer create-project laravel/laravel sample-project --prefer-dist
```

You can see sample-project represents whichever name you want to give to it. In the book, and I don't mean to be funny, we are going call it sample-project. However, it would be wise if you pick your own name. You will need a unique name later in the book when we create our Facebook login via Socialite.

From now on, wherever I refer to sample-project, just substitute in the name of your application.

Assuming your root path is like mine, that you are working with var/www, then once you've run the create-project command, it puts the fresh install of Laravel in the following path:

```
var/www/sample-project
```

There are a lot of files in the framework so this will take a minute or two to download depending on your connection.

Create Project in IDE

The next step is to create the project in your IDE. Since I use PHP Storm, I will give instructions for that. I might not be showing the current version, so yours might look different. You can see the futility of trying to provide documentation for the tools.

Anyway, you can use the IDE of your choice, including free ones like Netbeans and Eclipse. If you are unfamiliar with these, please go back to the previous chapter.

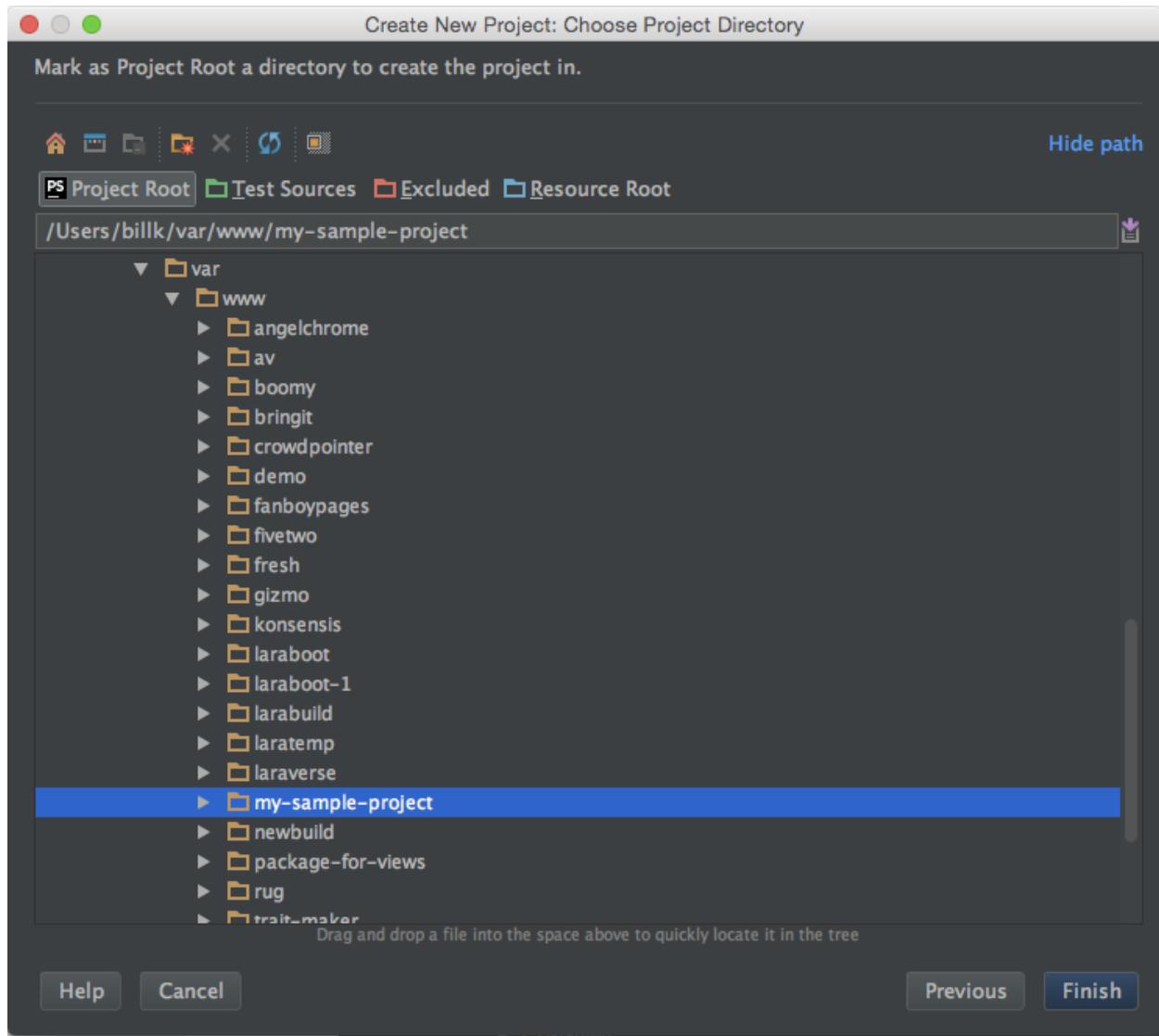
The main point is that you want to create the project from existing files:



On the next screen, if you are following along with PHP Storm, select the bottom option:

Source files are in a local directory, no Web server is yet configured

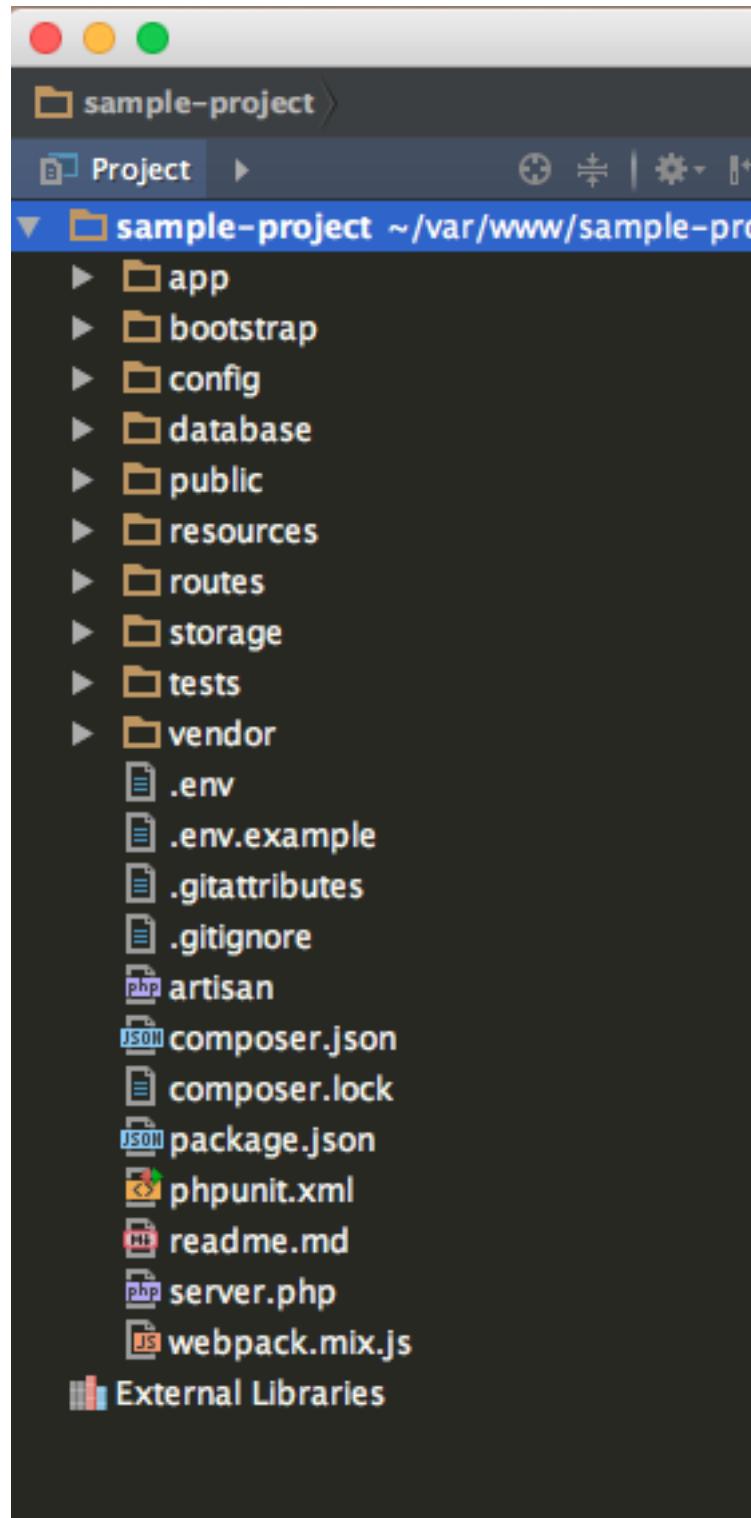
Next you will select the correct directory, in this case sample-project.



Select Folder

Please note the image says “my-sample-project”, but I’m actually using “sample-project.” Next click project root. Then select finish and PhpStorm will then set up your project.

You should then be able to see the following directory tree:



Ok, we're moving along. Now we need to be able to see our project in the browser.

Setup Local Host file:

Create local host entry for project name. We are going to use the .com extension, not .dev.

I had a lot of problems getting .dev to work correctly in my environment when I integrated Socialite, so I decided to switch to .com at this point. This stuff gets tricky, especially with the format of the callback url for facebook, which includes www.

You can stick with .dev if you really know your way around environments, but I found it really difficult to work with, so I just decided to make things easier on us. I also had to add a server alias for www as you will see in a moment.

To set up local host for a Mac, we will use vim to edit our hosts file. If you are on a mac, and you want to get to your root directory, type in the command line:

```
cd
```

Then from the command line, type the following:

```
sudo vim /private/etc/hosts
```

On a Mac, it will ask for your Mac password, so type that in and press enter.

Then type i for insert mode. Just to reiterate, use your-project-name.com, not sample-project.com for the following.

Scroll the cursor down and then create an entry for your host name, example:

```
127.0.0.1 sample-project.com www.sample-project.com
```

Then hit esc, then :wq to write and quit.

On Windows, open notepad in administrator mode and go to:

```
c:\Windows\System32\drivers\etc
```

select all Files for file types, then select:

```
hosts
```

Then add your-project-name.com to your hosts file:

```
127.0.0.1 sample-project.com www.sample-project.com
```

My instructions for Windows may be out of date, since I no longer use Windows. If they are not accurate, Google instructions for your version of Windows if you need them.

Vhost Entry

Next we need to create a vhost entry inside MAMP. The path on a Mac that we are looking for is:

Applications/MAMP/conf/apache/extra/httpd-vhosts.conf

Add the following entry:

```
<VirtualHost *:80>
    ServerAdmin webmaster@dummy-host2.example.com
    DocumentRoot "/Users/billk/var/www/sample-project/public"
    ServerName sample-project.com
    # available aliases to use
    ServerAlias www.sample-project.com
    ErrorLog "logs/dummy-host2.example.com-error_log"
    CustomLog "logs/dummy-host2.example.com-access_log" common
</VirtualHost>
```

Note that the path for document root is specific to my computer, adjust yours accordingly.

Pay careful attention to the document root. You need to reference the public folder. And as a last reminder, substitute in your-project-name for sample-project.

If you are doing this on Windows, it's pretty much the same, but your document root should look something like:

```
DocumentRoot "C:\var\www\sample-project\public"
```

For Windows, note the use of backslashes. This assumes you are using a var directory on your C drive. Adjust your path accordingly if it is different.

httpd.conf

Find your Apache httpd.conf file, which, on my mac is here:

/Applications/MAMP/conf/apache/httpd.conf

Make sure in the apache httpd.conf file to uncomment the following, so it looks like this:

```
# Virtual hosts
Include /Applications/MAMP/conf/apache/extra/httpd-vhosts.conf
```

Again, this is for a mac install where MAMP is in the Applications folder. For Windows users, find httpd.conf from wherever you installed your MAMP folder.

What this does is allows the conf file to include the vhosts, which we made an entry for sample-project.

MOD Rewrite

In the httpd.conf file, check to see if the rewrite module is enabled. It should be uncommented and look like this:

```
LoadModule rewrite_module modules/mod_rewrite.so
```

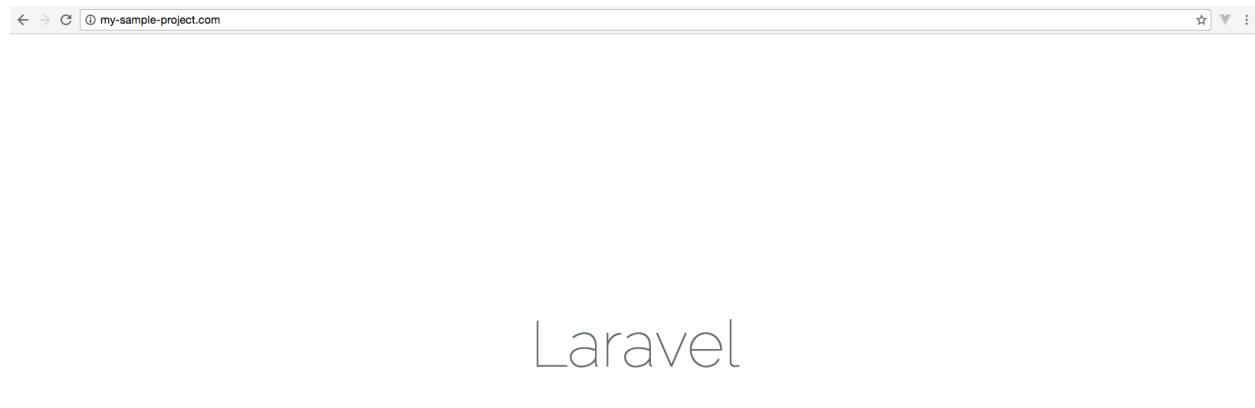
This allows us to display pretty urls, urls without index.php.

If you have problems with pretty urls resolving, consult the [Laravel Docs](#) for an alternative, and if that doesn't solve it, try Google or Stackoverflow.

Restart Apache

Next we restart Apache, so if you already had it running on MAMP, go ahead and restart it.

Now when you type in sample-project.com into the browser, you should get the basic Laravel page:



You can check your version of Laravel you installed by going to the command line and running:

```
php artisan --version
```

Summary

Congratulations, the most difficult part of the book is over. Hopefully it went smoothly for you. If you are not seeing the above result, then go back over the instructions carefully until you find where it is breaking.

I know it can be difficult when you are just learning. It takes a lot of work to set up a proper development environment.

At the same time, I feel compelled to point out that Laravel itself was the easiest install we had to work with. It was just one Composer command. Can't get much simpler than that.

Chapter 4: Let's Get Started With Laravel

Since this book is helping beginners, I will briefly mention setting up a repository and version control via Git. This is optional, you don't have to do it to follow along with the book, but it is a best practice. Obviously more advanced users already follow these practices.

Set Up The Repository

Most developers use a 3rd party hosting company to keep a copy of their work. If your computer melts, your project is backed up, and you also have a common point of reference for collaboration. Repositories are widely used by most developers.

Two of the big repository hosts are Github and BitBucket. Github charges for private repositories and Bitbucket is free. Github is more or less the standard, but then again Bitbucket is free. Free is great for learning and saving money. Currently, I'm using BitBucket for private work. The [public repository for Laraboot](#) is hosted on Github.

A quick note on the Github repository for this book. It is the final application, so when you are following along in the book, it's best to use the code I am providing within the context of the chapter it is written in.

It's beyond the scope of this book to go over the details of setting up a repository, but this is something I would encourage you to do on your own. Like I said, I'm using both Bitbucket and Github and they are both really easy to work with. Here are the links if you need them:

[BitBucket](#)

[Github](#)

Initial Commit

If you decided to install Git, which is versioning software,not to be confused with Github, which is repository hosting, then now you could do an initial commit and push it to your repository.

Again, we don't really cover Git in this book, it's something you need to explore on your own if you are not already familiar with it.

Obviously with large and complex code bases, being able to manage versioning is critical. Otherwise it's too easy to turn your magnificent work of architecture into a pile of rubble.

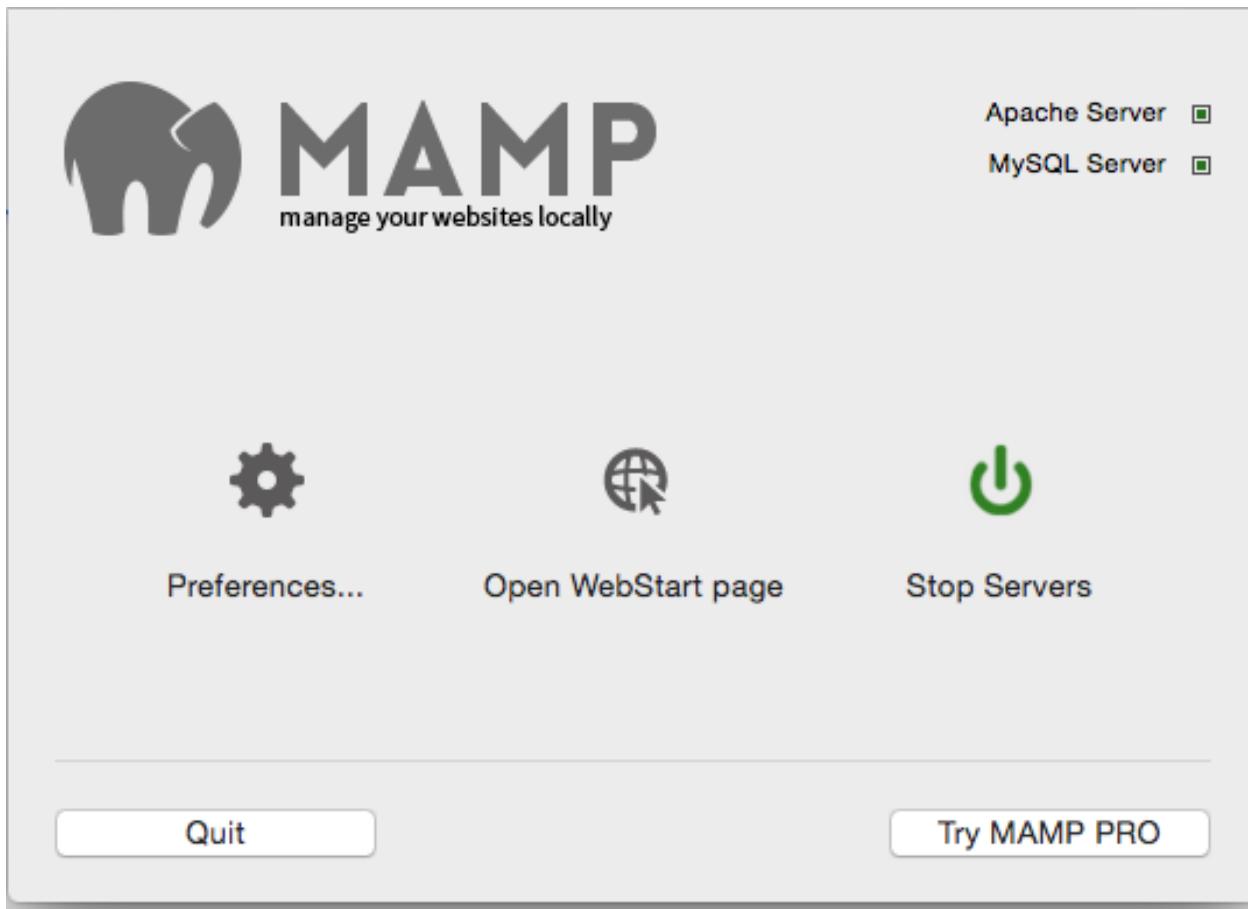
If you need the link for Git, I provided it in the 2nd chapter, but here it is again:

[Git](#)

Diving Into Workflow

I think the best way to learn Laravel is to dive right into workflow. When I'm building an application, the first thing I do is setup the database, henceforth known as the DB.

To do that, we need to make a quick side trip over to Php MyAdmin via MAMP. Select the MAMP start page from the MAMP application window:



Please note the above image is from my last install of MAMP, it's been a while, so your version may look different.

Click on open WebStart page, and then you will see:

PHP

[phpinfo](#) shows the current configuration of PHP.

MySQL

MySQL can be administered with [phpMyAdmin](#).

To connect to the MySQL server from your own scripts use the following connection parameters:

Host	localhost
Port	3306
User	root
Password	
Socket	/Applications/MAMP/tmp/mysql/mysql.sock

Examples

[PHP <= 5.5.x](#)

[PHP >= 5.6.x](#)

[Python](#)

[Perl](#)

mamp-my-page

You can see this going to show your MySQL info, including password, but of course I have deleted out my password. If your password is blank, then no password is set. You can set your password via the phpMyAdmin interface.

We need to look at the interface anyway because we have to create our database.

Setup the DB

The blue link for phpMyAdmin will take you to:

The screenshot shows the phpMyAdmin interface with the following details:

- Header:** Shows "Start", "My Website", "phpInfo", "Tools", "FAQ", and "MAMP Website".
- Left Sidebar:** Shows a tree view of databases: "New", "information_schema", "larabuild", "mysql", "performance_schema", "verse", and "yii2start".
- General Settings:**
 - Server connection collation: utf8mb4_unicode_ci
- Appearance Settings:**
 - Language: English
 - Theme: Original
 - Font size: 82%
- Right Sidebar:** Includes sections for "Database", "Web server", "phpMyAdmin", and "Help".

PHPMyAdmin

Select the databases tab. Then enter the db name, in your case, your-project-name. Next to that, we select the collation. We use utf8_unicode_ci. The reason we use that is so we get proper sorts.

This is what it looks like:

The screenshot shows the "Databases" tab in phpMyAdmin with the following details:

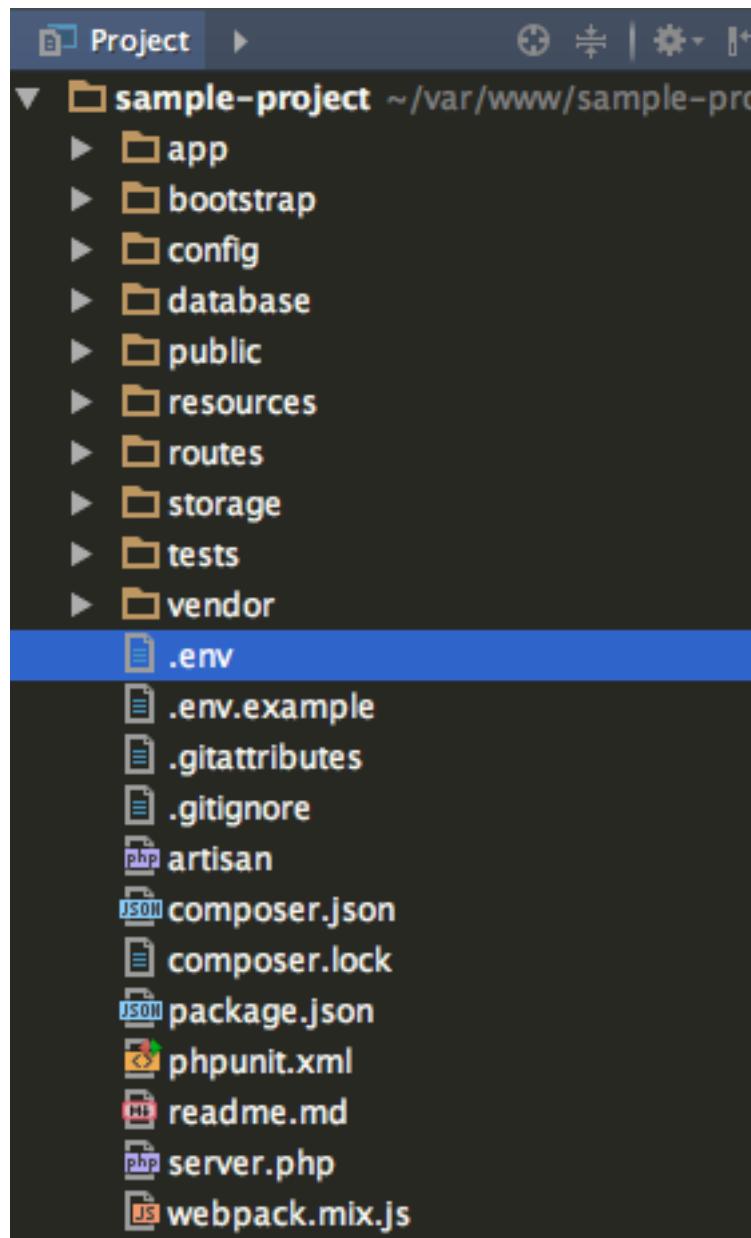
- Header:** Shows "Server: localhost:3306" and tabs for "Databases", "SQL", "Status", "Users", "Export", and "Import".
- Databases Section:** Shows a "Create database" button with a question mark icon. A text input field contains "sample-project" and a dropdown menu shows "utf8_unicode_ci".
- Footer:** Shows the word "databases".

Hit the create button and you have a DB named your-project-name, in my case, sample-project.

.env

Now we need our Laravel application to be able to communicate with our DB.

We do this by modifying the .env file. You can locate that file directly below, but not in, the vendor folder in your directory tree:



Let's look at what we get out of the box:

```
APP_ENV=local
APP_KEY=base64:your-random-string
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
QUEUE_DRIVER=sync

REDIS_HOST=127.0.0.1
REDIS_PASSWORD=null
REDIS_PORT=6379

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

PUSHER_APP_ID=
PUSHER_APP_KEY=
PUSHER_APP_SECRET=
```

Right away, we can feel the intuitiveness of Laravel. The .env file is for setting environment variables, easy enough to understand.

You can see that APP_ENV is set to local and APP_DEBUG is set to true, which means we will get helpful error messages.

When you use Composer to install Laravel as we did, the app key is automatically set for you. I've used YourRandomString in the example above, but obviously you will see your app key. This insures session and other encrypted data is secure.

The main thing we are going to do here is set our DB connection. Let's change it to the following:

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=sample-project
DB_USERNAME=root
DB_PASSWORD=your-password
```

Obviously in the above, you will be changing your-password to your actual password. Save the file and you now have a working connection to your DB.

We're going to use that a little later to hold our user records, so we can store the username and passwords when users register on our application. But before we build that, let's get a basic idea of how things work.

Application Structure

If you have taken a look at Laravel's folder structure, you may be a bit surprised. It's a fairly complicated structure with lots of folders. These folders are all perfectly logical, for example the app/Http folder houses the controllers, middleware, and kernel files. They all have to do with processing a request from the browser.

The kernel and middleware aspects are likely to be new to you, as they were to me when I first started with Laravel. Don't let it put you off, we will cover these in detail at the appropriate time, and you will see just how cool and innovative this framework is. And even though it's cool and innovative, that doesn't come at the expense of usability. These things actually make it easier to use.

But it does take a little getting used to before you remember where everything is, one reason why I'm not just going to rattle through all the folders and explain everything. You would never remember it.

It's better to learn it through workflow and practical application. And what you'll find is that the concepts encapsulated by the folders make more sense in the context of what you are trying to build. The reason for this is that it does such an awesome job of code separation and separation of responsibilities that it makes it easy to grasp the purpose of the classes.

But let's not jump too far ahead. Let's just take it one small step at a time.

Basic Stitching

To get a basic idea of how things stitch together in Laravel, we are going to build a test route, controller, and view, which will come in handy later in the book. Any time we want to play with code, we will have a test controller and view for that purpose. For now we'll use it to understand the connection between routes, controllers, and views.

In Laravel, every path through the application is defined by a route. Typically the route is going to point to a controller and the controller is going to return a view. It creates an endpoint for the application.

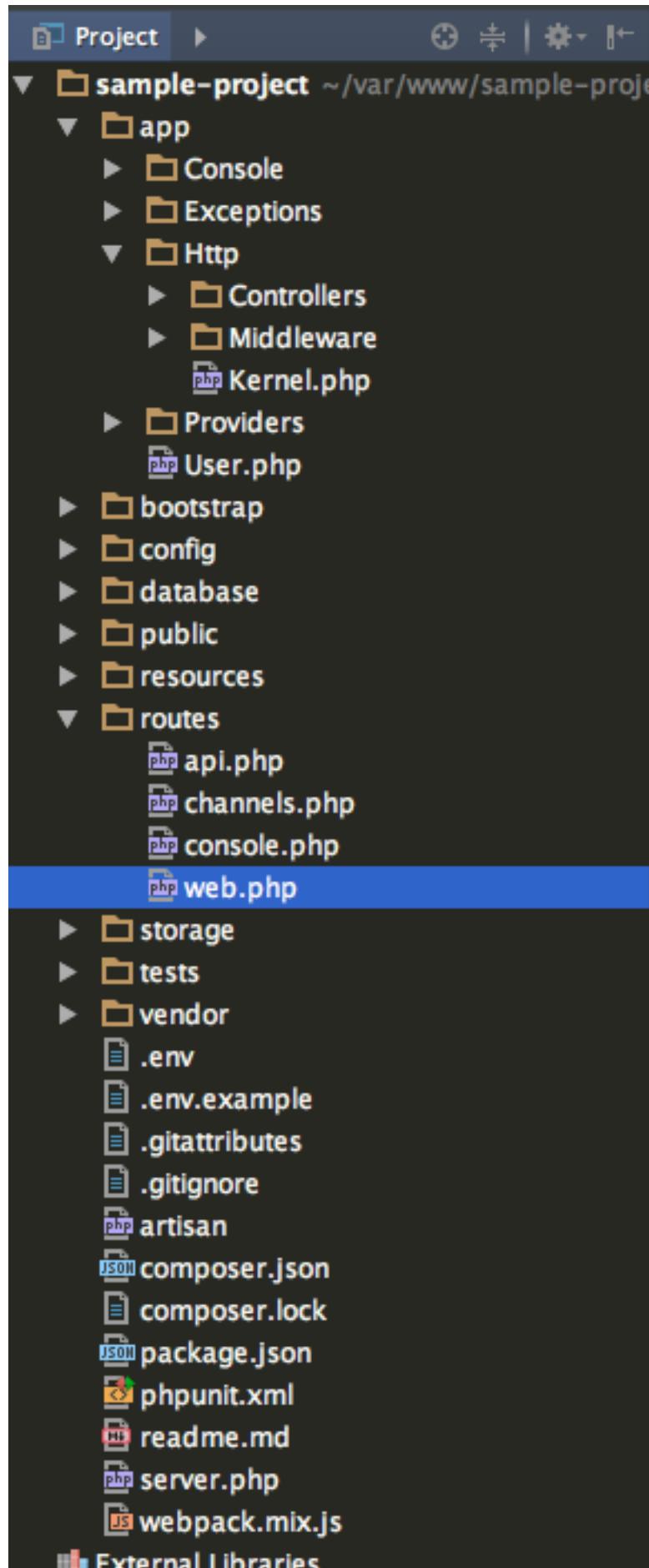
Routes

Routes have their own folder, and are divided into 4 files, api, channels, console, and web.

Let's start by looking at the web route file. Go to the following file and open it:

`sample-project/routes/web.php`

It's located here:



Here's the default route we get in the file:

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

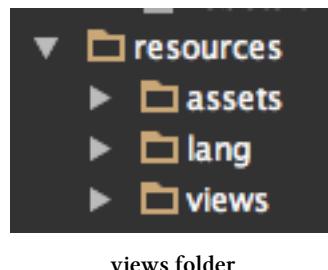
So what we have here is the get method of the Route class, with 2 parameters, the first is the uri “/”, which is the base url matching our root domain sample-project.com. It covers instances where there is backslash at the end and also without it, just the plain domain.

The second parameter is a closure, which is an anonymous function that immediately executes. In this case it's a function that returns a welcome view by using the view method and indicating which view we want in the method signature:

```
view('welcome');
```

This is what returns the welcome view we see when we type sample-project.com into the browser. Instead of pointing to a controller, it uses a closure to return the view.

Let's take a quick look at the view code. Find the resources folder and within that, find the views folder. It looks like this:



Inside the views folder, you will find welcome.blade.php. Here is the full path in case you need it:

app/resources/views/welcome.blade.php

Open the file so you can see what you get out of the box:

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

        <title>Laravel</title>

        <!-- Fonts -->
        <link href="https://fonts.googleapis.com/css?family=Raleway:100,600" rel="stylesheet" type="text/css">

        <!-- Styles -->
        <style>
            html, body {
                background-color: #fff;
                color: #636b6f;
                font-family: 'Raleway';
                font-weight: 100;
                height: 100vh;
                margin: 0;
            }

            .full-height {
                height: 100vh;
            }

            .flex-center {
                align-items: center;
                display: flex;
                justify-content: center;
            }

            .position-ref {
                position: relative;
            }

            .top-right {
                position: absolute;
                right: 10px;
                top: 18px;
            }
        </style>
    </head>
    <body class="flex-center full-height position-ref">
        <div class="top-right" style="background-color: black; color: white; padding: 5px; border-radius: 50%; width: 20px; height: 20px; display: flex; align-items: center; justify-content: center;">4</div>
    </body>
</html>
```

```
.content {
    text-align: center;
}

.title {
    font-size: 84px;
}

.links > a {
    color: #636b6f;
    padding: 0 25px;
    font-size: 12px;
    font-weight: 600;
    letter-spacing: .1rem;
    text-decoration: none;
    text-transform: uppercase;
}

.m-b-md {
    margin-bottom: 30px;
}

</style>
</head>
<body>
    <div class="flex-center position-ref full-height">
        @if (Route::has('login'))
            <div class="top-right links">
                <a href="{{ url('/login') }}>Login</a>
                <a href="{{ url('/register') }}>Register</a>
            </div>
        @endif

        <div class="content">
            <div class="title m-b-md">
                Laravel
            </div>

            <div class="links">
                <a href="https://laravel.com/docs">Documentation</a>
                <a href="https://laracasts.com">Laracasts</a>
                <a href="https://laravel-news.com">News</a>
                <a href="https://forge.laravel.com">Forge</a>
                <a href="https://github.com/laravel/laravel">GitHub</a>
            </div>
        </div>
    </div>
</body>
```

```
</div>
</div>
</body>
</html>
```

The Style Problem

Before we go further, I need to bring up a point about how the book is formatted. In the above code snippet, you can see:

```
<link href="https://fonts.googleapis.com/css?family=Lato:100" rel="stylesheet" type="text/css">
```

If you look closely, you may see a nasty backslash, depending on how you view this page, that does not belong there in the middle of the word type. This is what happens when the code word-wraps and this will absolutely break the code.

I'm 100% committed to giving you working code. I want you to be able to just copy and paste it. But that means I have to format the code in a way that it makes the code stylistically less appealing, to say the least.

For example, to avoid the breaking backslash in the above snippet, I could do it like this:

```
<link href="https://fonts.googleapis.com/css?family=Lato:100"
      rel="stylesheet"
      type="text/css">
```

So I will be adjusting the code to account for word-wraps, and I will add spaces for readability. I will also be providing Gists, which are hosted snippets of code on Github that you can copy freely, on most code of any significant length. You should use those when you can.

Ok, let's get back to talking about the welcome.blade.php file. Even though it's a blade file, which we know because of the file convention welcome.blade.php, it is not using any of blade's amazing syntax.

Instead it's just some raw html and css that gets us our welcome page. What a perfect starting point for us because you can see there is nothing special about it, it just puts "Laravel" on the page, along with some links.

So just to recap, when you type sample-project.com in the browser url bar, it calls the route "/" and returns the view('welcome'), whose markup you see in welcome.blade.php. Pretty simple stuff.

Unit Testing

As a nice little tie-in, we can actually test our route via php unit test.

Right out of the box, Laravel comes with unit testing already set up. If you open sample-project/tests/Feature/ExampleTest.php, you will see the following:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

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

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

It's a simple test on the route to see if it will return status 200, which is success, for the route, which we know it should.

To run this test, all we have to do is type the following from the command line and hit enter:

```
vendor/bin/phpunit
```

You should get a response like this:



```
Bills-MacBook-Pro:~ billk$ cd var/www/sample-project
Bills-MacBook-Pro:sample-project billk$ vendor/bin/phpunit
PHPUnit 5.7.6 by Sebastian Bergmann and contributors.

..
2 / 2 (100%)

Time: 147 ms, Memory: 12.25MB

OK (2 tests, 2 assertions)
Bills-MacBook-Pro:sample-project billk$
```

So obviously that tests the response we expect to see from our route. It also runs a sample test in the sample-project/tests/Feature/ folder, which is just a stub asserting true.

Pretty cool stuff. Laravel makes this so easy for you. It's all set up and ready to go out of the box. Just add tests to the Unit or Feature folder as you wish and run the same command and it will run all the tests for you.

Ok, so back to the basics. So far the route in the test goes directly to a view, but that's not typical. More often than not, a route will go to a controller, so our route will be a little different.

Luckily the syntax is even more intuitive when using a route.

Creating a Route

As we go through fitting together the pieces of the puzzle, the route, controller, and view, we will keep an eye towards developing a workflow that we can standardize. Doing this helps us visualize the process, so when we add a feature to our application, we know the steps we are going to take to implement it and in what order.

Routes are a good place to start. How will the user traverse through the site? In order for someone to land somewhere in your application, there has to be a route to take them there.

Laravel has a lot of robust routing features. I'm not going to list them here because the [laravel docs](#) do a great job of listing everything. So instead, I'll just introduce you to features as we use them through workflow.

Laravel ships with the following route defined:

```
// homepage route

Route::get('/', function () {

    return view('welcome');

});
```

I put the comment above the route. Ok, let's create a route for our test controller and view. Ultimately, we want our url to look like this:

sample-project.com/test

We want that url to resolve to an index method on our controller, and within the index method, we are going to tell it to return the index view.

Ok, let's add the following to sample-project/routes/web.php:

```
// homepage route

Route::get('/', function () {

    return view('welcome');

});

// test route

Route::get('test', 'TestController@index');
```

Note that the get method corresponds to the type of request. This is literally a GET request. Makes it fairly intuitive and easy to guess what a POST request would look like. For example, if you had a route from a create form that was going to post the data:

```
Route::post('test', 'TestController@store');
```

Even though the first parameter is the same, Laravel knows it's a different route because of the method, it's a POST, not a GET. Anyway, don't add that route, we don't need it, I just mentioned it for an example.

Also note that I'm placing the routes in order of the comments in alphabetical order. That's why I put a comment above the route, so I can easily identify the route. This comes in very handy when you have a lot of routes in your application.

So now we have our get route, but we don't have a corresponding controller.

Creating a Controller

In workflow, we jump around a lot, it's the nature of the game. To create our controller, we're going to use Artisan, laravel's amazing command line tool.

Artisan

If you want to see the list of available commands for Artisan, type the following in the command line and press enter:

```
php artisan list
```

That will return the list of available commands:

```
app
  app:name      Set the application namespace
auth
  auth:clear-resets Flush expired password reset tokens
cache
  cache:clear    Flush the application cache
  cache:table   Create a migration for the cache database table
config
  config:cache  Create a cache file for faster configuration loading
  config:clear  Remove the configuration cache file
db
  db:seed       Seed the database with records
event
  event:generate Generate the missing events and listeners based on registration
key
  key:generate Set the application key
make
  make:auth     Scaffold basic login and registration views and routes
  make:command  Create a new Artisan command
  make:controller Create a new controller class
  make:event    Create a new event class
  make:job      Create a new job class
  make:listener Create a new event listener class
  make:mail     Create a new email class
  make:middleware Create a new middleware class
  make:migration Create a new migration file
  make:model   Create a new Eloquent model class
  make:notification Create a new notification class
  make:policy   Create a new policy class
  make:provider Create a new service provider class
  make:request Create a new form request class
  make:seeder   Create a new seeder class
  make:test    Create a new test class
migrate
  migrate:install Create the migration repository
  migrate:refresh Reset and re-run all migrations
  migrate:reset  Rollback all database migrations
  migrate:rollback Rollback the last database migration
  migrate:status Show the status of each migration
notifications
  notifications:table Create a migration for the notifications table
queue
  queue:failed   List all of the failed queue jobs
  queue:failed-table Create a migration for the failed queue jobs database table
  queue:flush    Flush all of the failed queue jobs
  queue:forget   Delete a failed queue job
  queue:listen  Listen to a given queue
  queue:restart Restart queue worker daemons after their current job
  queue:retry   Retry a failed queue job
  queue:table   Create a migration for the queue jobs database table
  queue:work    Process the next job on a queue
route
  route:cache  Create a route cache file for faster route registration
  route:clear  Remove the route cache file
  route:list   List all registered routes
```

Obviously, I couldn't fit the entire list in the image. If the image is not clear in your device format, just run the command and you can see the list more clearly from the command line.

You can see Artisan does quite a lot, much more than we can discuss at the moment and more than you will remember in a single look. The thing we are interested in now is using Artisan's ability to make a class for us, like a controller. So let's just focus on that one for now.

Go to your command line and type in the following command and hit enter:

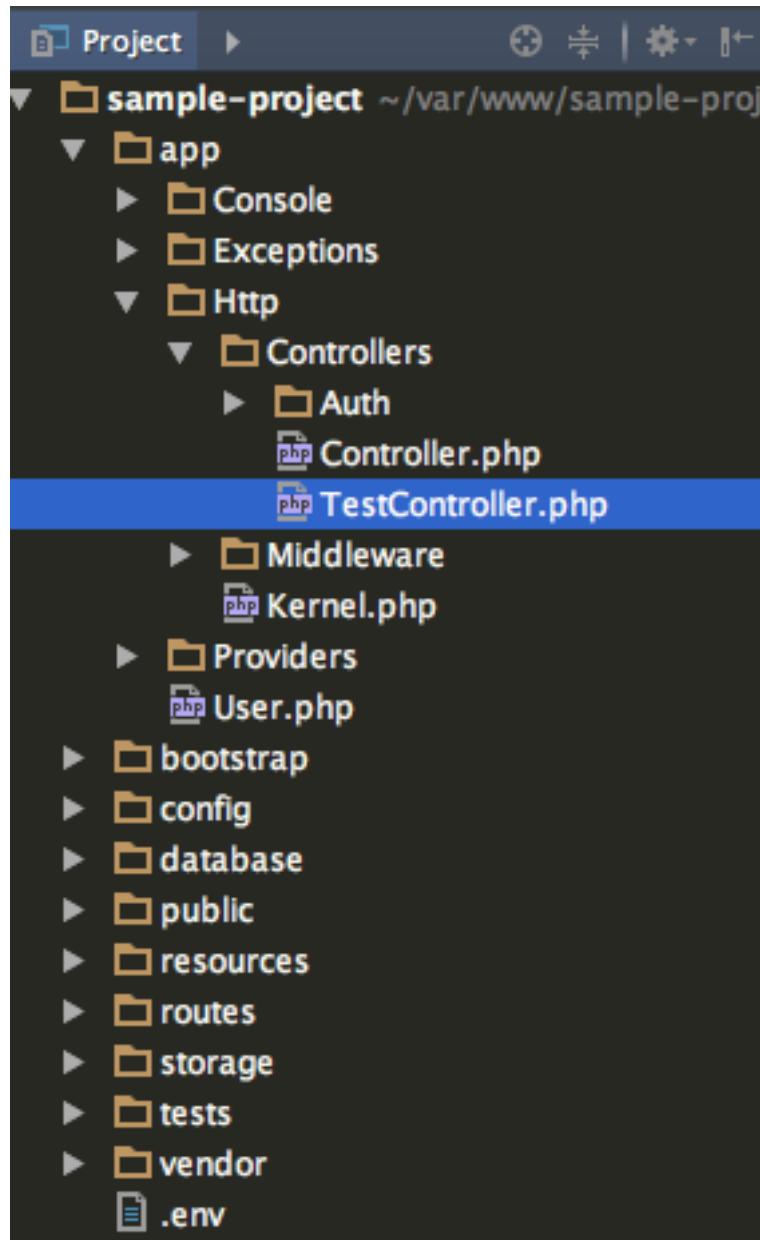
```
php artisan make:controller TestController
```

This just two simple parts, the command and the name of the class. You can see the convention is to include the word controller and uppercase on the first letter of all the words in the name of the controller.

After pressing enter, you should get a confirmation message like this:

```
Bills-MacBook-Pro:my-sample-project billk$ php artisan make:controller TestController
Controller created successfully.
Bills-MacBook-Pro:my-sample-project billk$ █
```

Now if you go look in the following folder, sample-project/app/Http/Controllers, you will see the following TestController file:



This is the code in the file:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class TestController extends Controller
{
    //
}
```

So you can see we mostly got was an empty stub, with no methods. Laravel was smart enough to namespace it properly for us and place it in the correct folder.

Go ahead and delete that file. We are going to create the test controller again, but this time, we are going to use the `--resource` flag, so after deleting, type in the following in the command line:

```
php artisan make:controller TestController --resource
```

We got 7 empty methods that follow the RESTful pattern:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class TestController extends Controller
{
    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        //
    }
}
```

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

/**
 * Store a newly created resource in storage.
 *
 * @param \Illuminate\Http\Request $request
 * @return \Illuminate\Http\Response
 */
public function store(Request $request)
{
    //
}

/**
 * Display the specified resource.
 *
 * @param int $id
 * @return \Illuminate\Http\Response
 */
public function show($id)
{
    //
}

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

/**
```

```
* Update the specified resource in storage.  
*  
* @param \Illuminate\Http\Request $request  
* @param int $id  
* @return \Illuminate\Http\Response  
*/  
public function update(Request $request, $id)  
{  
    //  
}  
  
/**  
 * Remove the specified resource from storage.  
 *  
 * @param int $id  
* @return \Illuminate\Http\Response  
*/  
public function destroy($id)  
{  
    //  
}  
}
```

RESTful pattern

When I first started learning, I didn't know what RESTful meant, so I looked it up and came across the wikipedia article:

[REST: Representational state transfer](#)

From the Article:

“To the extent that systems conform to the constraints of REST they can be called RESTful. RESTful systems typically, but not always, communicate over Hypertext Transfer Protocol (HTTP) with the same HTTP verbs (GET, POST, PUT, DELETE, etc.) that web browsers use to retrieve web pages and to send data to remote servers.”

Ok, we can see it involves the HTTP verbs, even if it doesn't completely explain what it is. Anyway, it doesn't matter. RESTful in the way we use it with Laravel is a pattern and that's easy and clear enough to see.

7 RESTful routes match 7 RESTful controller methods. We will see this in action later when we create a route resource, which will automatically create the RESTful routes matching to the RESTful controller methods. You will love how easy it is.

For now, let's look at the TestController that we created. We have our PHP opening tag, and just as a note, we do not use closing php tags in our classes, they are not necessary and will cause problems, so they are

intentionally omitted. You probably already knew that, but I mentioned it just for the sake of anyone who might not know it.

Namespaces and Use Statements

Next we get our namespace declaration, followed by a use statement:

```
namespace App\Http\Controllers;  
  
use Illuminate\Http\Request;
```

You can see that we are following [PSR-4](#) namespace convention, where the top level is App. This is very logical, it simply follows the folder structure of the application. TestController.php resides in the Controllers directory, so you can see how this works, if you just follow the path of the folders.

Use statements are similar in that they follow the folder structure. The difference is that the last part is the name of the actual class.

One thing I always hated about technical books is when they fail to mention the use statements that are needed to run a block of code. Most often, it's easy enough to figure out when a use statement is missing, but not always. In some cases, like with exception handling, if the use statement is missing, Laravel will not return an error and you won't know why. If that happens, make sure you have all the required use statements that you need to run the code. I'll do my best to make sure they are always included in my instructions.

The Request class that starts with Illuminate is deep in the framework. You can find it here:

```
vendor/laravel/framework/src/illuminate/Http/Request.php
```

If you open that file, you can see it's namespaced as follows:

```
namespace Illuminate\Http;
```

Note the use of the Illuminate namespace, which means it's framework class found in vendor/laravel/framework/src/illuminate.

We don't need to worry about the contents of that file, I just wanted to show you where the illuminate files reside, if you want to reference them.

Back to our controller. Next in our file, we have we have the class declaration:

```
class TestController extends Controller
{
```

Simple enough. Since all the methods of the class are empty stubs, nothing to discuss there, except the restful pattern, which you can see there is a method for each one:

- index
- create
- store
- show
- edit
- update
- destroy

We will see this pattern in detail in a later chapter, so I will leave it at that for now.

Index Method

Let's modify the index method to the following:

```
public function index()
{
    return 'made it here.';
}
```

So now when you visit:

`sample-project.com/test`

You see:

made it here.

We don't want this of course, we are just testing our way forward to make sure our route is working. Let's modify the index method to:

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

We are telling it to look for a view folder named test in the views directory and within that folder find a view named index.blade.php. You can use a forward slash between the folder and view if you wish, but I love the dot notation.

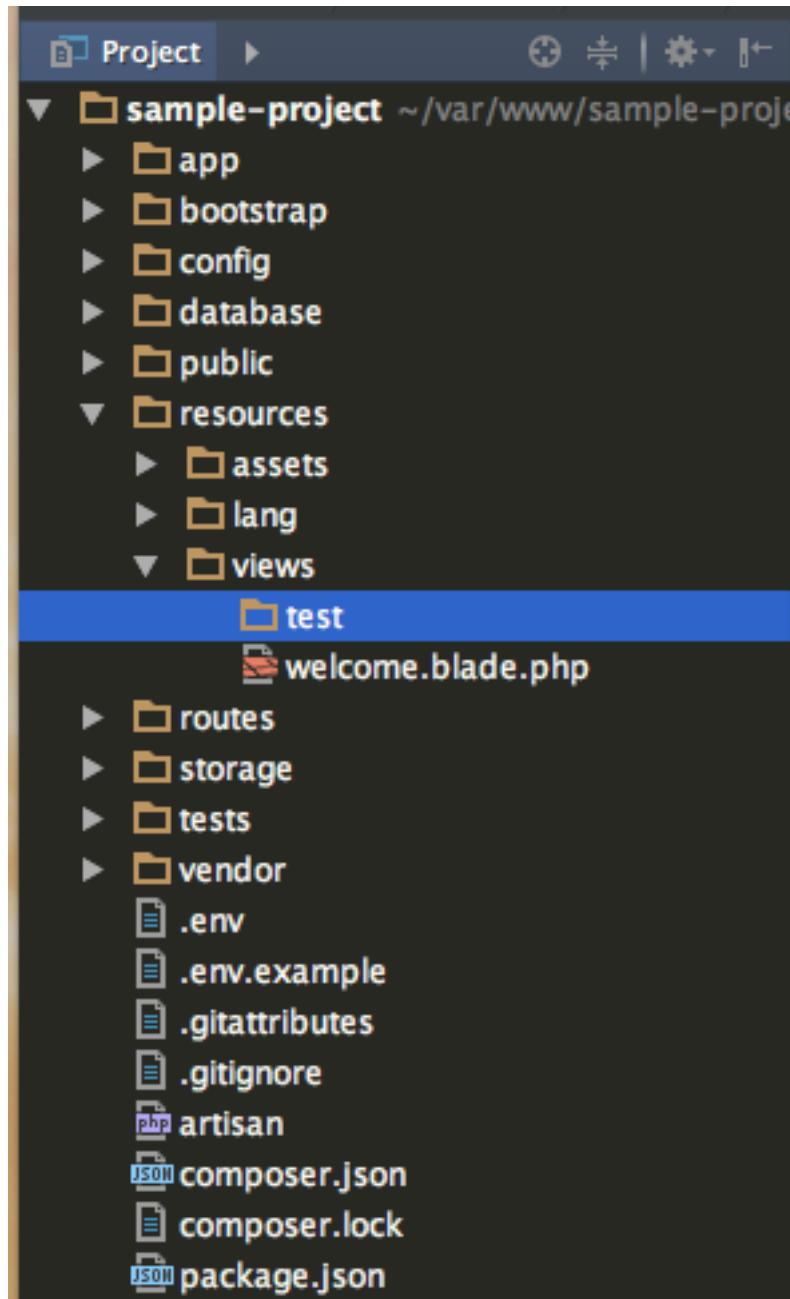
With Laravel, as with many frameworks, there is often more than one way to do something. In this book, I'm going to focus on specific implementations that follow the way I tend to do things.

This will stop the book from turning into just another version of the docs, and otherwise we will just drown in too much information. That said, it would be wise to read through the docs at your leisure to see all the variations and interesting things you can do. I would highly recommend that.

Ok, back to the view. Now obviously 'test.index' won't work because this file does not exist.

Views

Let's make our view now. Go to sample-project/resources/views and create a new folder inside the views folder named test. It should look like this:



Now inside the test folder, create a new file named index.blade.php. If your IDE placed any boilerplate PHP, strip it out until you have a blank page and then just put the following in the index.blade.php file:

```
<h1>This is My Test Page</h1>
```

Even though I didn't declare any html tags, I'm using google Chrome as my browser and it compensates, so you'll see 'This is My Test Page' when you visit:

```
sample-project.com/test
```

Now don't worry about the missing html markup, I left it out for a reason. First we need to learn a little about Blade, Laravel's templating engine before moving on.

Blade

You already know that we declare a blade template by using the convention filename.blade.php. Laravel reads this, then compiles Blade's template language to PHP. Because of this, there is no overhead for using Blade. Awesome!

We're going to do a gentle introduction to Blade's syntax with if and foreach statements. We will be using these throughout the book and you will see just how useful and how great Blade is to work with.

Ok, let's go back to the TestController and change the index method to the following:

```
public function index()
{
    $beatles = ['John', 'Paul', 'George', 'Ringo'];

    return view('test.index', compact('beatles'));
}
```

The beatles were a music group from the 1960s in case you don't know. They were so famous they were known by their first names. Decades later, they still have fans, and I'm one of them.

Anyway, what we're doing is taking our \$beatles array and passing it along to the view via the compact method. The compact method is native PHP and will allow us to reference \$beatles array in our view page.

Just a note. The compact method can accept multiple variables, so if you had multiple items you wanted to pass along, for example:

```
return view('test.index', compact('beatles', 'stones', 'zeppelin'));
```

All of those would be available to the view, provided there were matching variables in the index method. But for now we'll keep our example extremely simple and just stick to beatles.

So let's go to our index.blade.php file and add the following:

```
@foreach($beatles as $beatle)  
{{ $beatle }}  
@endforeach
```

We use @foreach to declare the foreach loop, no need to add any php to it. Then we use the {{ brackets to open and close the php echo statement. Then we close it up with @endforeach.

You can see how clearly the html and blade syntax separate from each other. No messy PHP declarations needed. It's incredibly easy to read. This is so wonderful to work with, you will really appreciate this by the time we get done with this book.

Let's imagine we only want to display the results of an object if there are some to display. In this case, we'll add an if statement:

```
<h1>This is My Test Page</h1>  
  
@if(count($beatles) > 0)  
@foreach($beatles as $beatle)  
{{ $beatle }}  
@endforeach  
  
@else  
  
<h1> Sorry, nothing to show... </h1>  
@endif
```

We're just using PHP's count function to see if we have anything to iterate through, and if so run the foreach to echo out the results, else echo out the 'Sorry, nothing to show...' statement. If you want to test the else statement, try this:

```
@if( ! count($beatles) > 0)
```

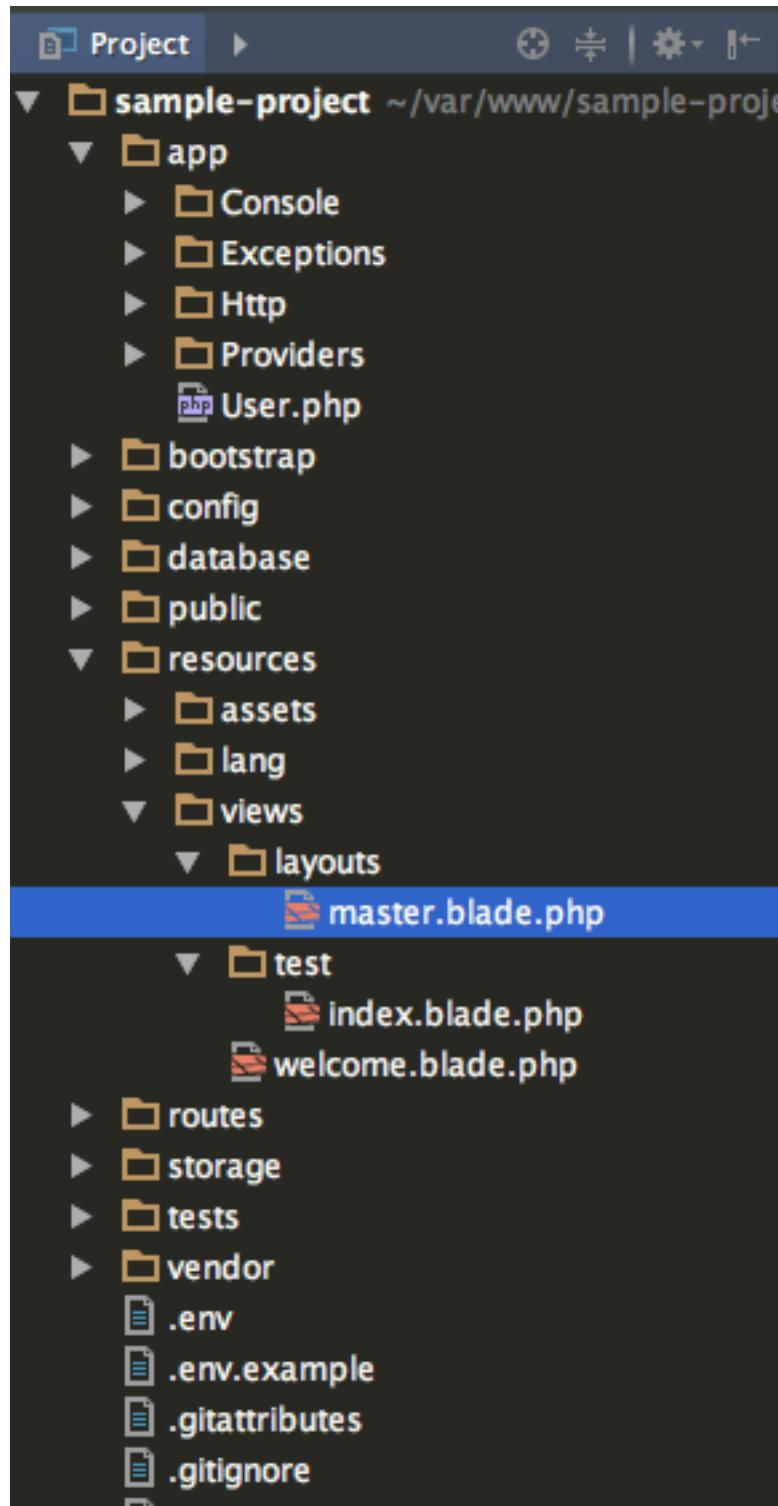
That will force the page to echo the else statement.

Ok, this is kind of fun. You can see how powerful this is. We haven't worked with models or objects yet, but don't worry, that's coming.

Creating A Master Page

Before we close out the chapter, we're going to set up what is known as a 'Master' page. What the master page does is hold the parts of the html that is repetitive, which then can be used by the individual pages. Let's start setting that up.

In the views folder, let's create a new folder named layouts. Within the layouts folder, let's create master.blade.php. It should look like this:



So what we want to do inside master.blade.php is to create a template page that holds the header and footer and injection points for the content in the body. You will see what I mean as we put it together.

First we need to grab some html to work with. Since we're building a reusable template, I'm actually going to be fairly thorough. We're going to use Twitter Bootstrap 3.

Bootstrap is widely used and for good reason. It's a powerful front-end framework that will allow us to stand up a mobile responsive page quickly. If you are not familiar with Bootstrap, you should really take the time to get to know it. You can find it here:

[getBootstrap.com](http://getbootstrap.com)

On that site, we're going to grab the source code to the following page:

[Bootstrap Theme](#)

So all I'm going to do is view source and copy, then paste the contents into master.blade.php.

It's too many lines to include the whole thing here, so I'm going to chop out what we don't need, make a few small tweaks to get us started, and show you what you want to put in master.blade.php.

Also, it's time to start using Gists. Like I said before, Gists are hosted code snippets on Github, which are super easy to copy and paste. The code goes directly from my IDE to the Gist, so I know it's working code. Here is the first one:

Gist:

[master.blade.php - first draft](#)

Obviously, you can still read along with the code in the book. Just be mindful of the accommodations I had to make to avoid wordwrap.

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

  <!-- The above 3 meta tags *must* come first in the head;
      any other head content must come *after* these tags -->

  <meta name="description" content="">
  <meta name="author" content="">

  <link rel="icon" href=".../favicon.ico">

  <title>Theme Template for Bootstrap</title>

  <!-- Styles -->
```

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

</head>

<body role="document">

<!-- Fixed navbar -->

<nav class="navbar navbar-inverse navbar-fixed-top">
  <div class="container">
    <div class="navbar-header">

      <button type="button"
        class="navbar-toggle collapsed"
        data-toggle="collapse"
        data-target="#navbar"
        aria-expanded="false"
        aria-controls="navbar">

        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>

      </button>

      <a class="navbar-brand"
        href="#">

        sample-project

      </a>

    </div>

    <div id="navbar"
      class="navbar-collapse collapse pull-right">

      <ul class="nav navbar-nav">

        <li class="active"><a href="#">Home</a></li>
        <li><a href="#about">About</a></li>
        <li><a href="#contact">Contact</a></li>

      </ul>

    </div>

  </div>

</body>
```

```
<li class="dropdown">
  <a href="#" class="dropdown-toggle" data-toggle="dropdown" role="button" aria-haspopup="true" aria-expanded="false">
    Dropdown <span class="caret"></span></a>

    <ul class="dropdown-menu">
      <li><a href="#">Action</a></li>
      <li><a href="#">Another action</a></li>
      <li><a href="#">Something else here</a></li>
      <li role="separator" class="divider"></li>
      <li class="dropdown-header">Nav header</li>
      <li><a href="#">Separated link</a></li>
      <li><a href="#">One more separated link</a></li>
    </ul>
  </li>

</ul>

</div><!--/.nav-collapse -->

</div>

</nav>

<div class="container theme-showcase" role="main">

  @yield('content')

</div> <!-- /container -->

<script src="/js/app.js"></script>

</body>
</html>
```

Ok, with that out of the way, let's talk about what I did differently from the source code on getbootstrap.com.

I want you to be able to repeat this process at will when you are looking to create sites and use different templates.

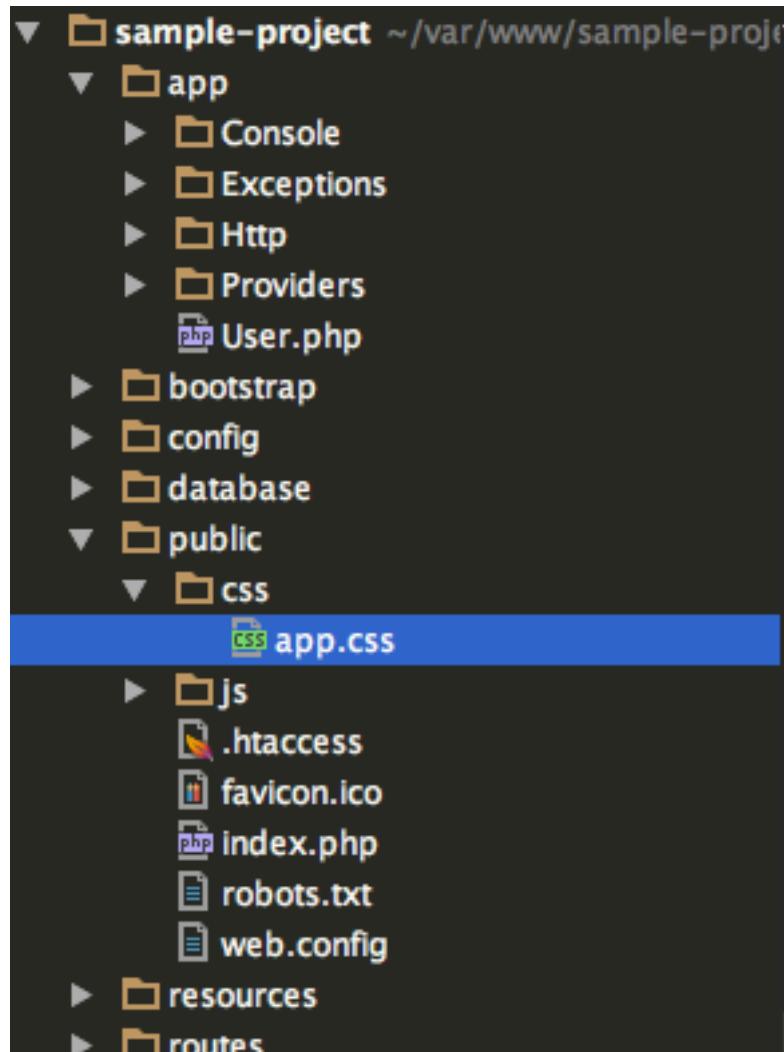
CDN

Many of you may be familiar with a CDN. I grabbed the code from the link I showed you, and part of that pulled in the necessary files via CDN. I have removed those calls.

A CDN is a content delivery network and it's an easy way to use the Bootstrap and jquery files that are often needed. There are many CDNs, just google for what you need.

This is how it works. A content provider provides a live feed for the files, which are then cached in your browser. Since they are so common, most users will have them cached in their browsers, so when the user hits your page, it loads instantly.

Unfortunately, this is not a robust solution. Too many CDN calls bog down the browser and there are limits to how many calls you make. Luckily for us, Laravel 5.4 ships with Bootstrap and Jquery included, so I dropped the CDN calls, and instead make calls to our local files, which are located in the public directory:



Note that we don't modify the css file or the js file directly, we do it via Laravel Mix, which compiles our assets into these files. I'm going to cover Mix later in the book, but we're not ready for that just yet.

So let's go back to master.blade.php and if you look at the code carefully, in the head section, you will see the following:

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

So that is the call to our css file.

And just before the closing body tag, you will see:

```
<script src="/js/app.js"></script>
```

Obviously, that pulls in our javascript file.

You'll note that I also added the following line in the body:

```
@yield('content')
```

This tells the master page to inject a section named content when the master page is extended by another page. We're going to try this with our test.index view, which should demonstrate the concept fully.

In our test.index view, which, just to be clear, is the index.blade.php file in the test folder, we need to add to change it to the following:

Gist: [test.index revised](#)

From book:

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

@section('content')

<h1>This is My Test Page</h1>

@if(count($beatles) > 0)

@foreach($beatles as $beatle)

{{ $beatle }}<br>

@endforeach
```

```
@else
```

```
<h1> Sorry, nothing to show...</h1>
```

```
@endif
```

```
@endsection
```

You can see the first line now is:

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

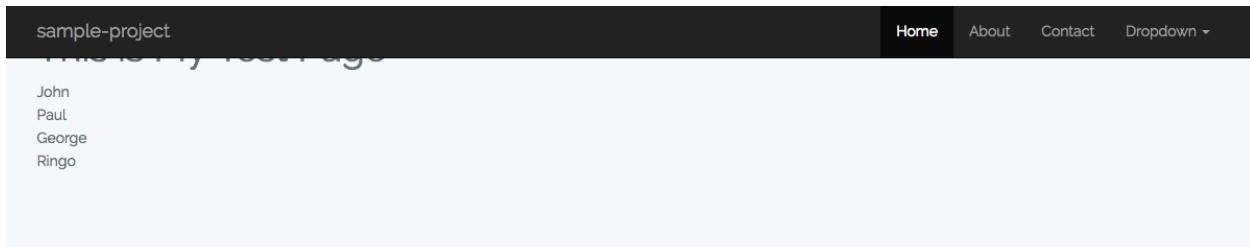
By extending the layouts.master page, we are going to inject the designated sections into it and render the composite of the individual view and the master page. Right now we only have one section named content. The tag for that is:

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

At the end of the file, you can see we are closing the section with:

```
@endsection
```

Now if you have all that working properly, you should be able to view the page at sample-project.com/test, however, there will be a problem:



You can see that the nav, because we are using a top-pin, is spilling over onto the body. Because of that we need write some css to handle it. Let's add a style tag within the header section of master.blade.php like so:

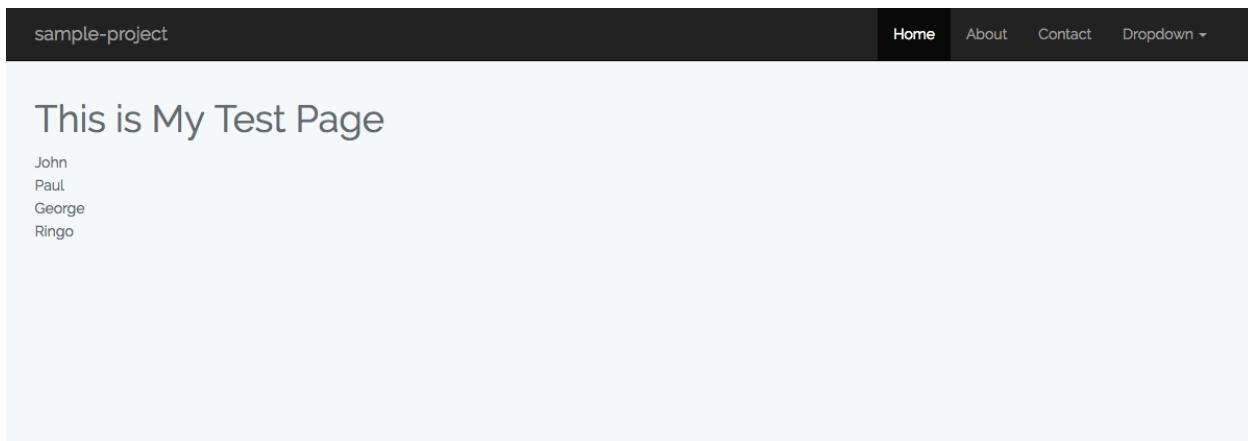
```
<!-- Styles -->

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

<style>
  body{
    padding-top: 65px;
  }
</style>
```

I included the call to the css file for reference, so obviously don't duplicate that.

So now, if you go to sample-project.com/test, you should see:



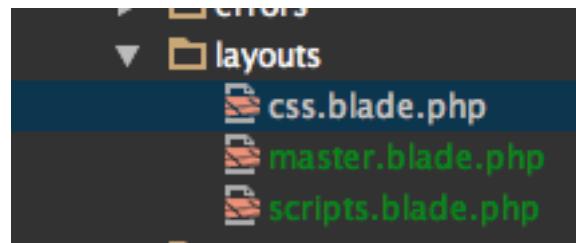
Ok, so we got our master page up and running, but there are still a few more tweaks we can do to help us with organization.

View Partials

The idea is to set up our master file so that it is incredibly easy to add to it and keep the code clean and readable. Right now we have too much in one file. To solve this, we are going to extract out some of the code into separate view files, which are often referred to as view partials.

Once we have extracted out the code, we pull it back in via Blade's @include method.

So in our views/layouts folder, let's create a css.blade.php file and a scripts.blade.php file. Your layouts folder should look like this:



Let's cut the following from master.blade.php and paste it into css.blade.php:

```
<!-- Styles -->

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

<style>

body{

padding-top: 65px;

}

</style>
```

Then in master.blade.php, where the css previously was, add the following line:

```
@include('layouts.css')
```

Now let's do the same for our javascript. Let's cut the following and paste it into scripts.blade.php:

```
<script src="/js/app.js"></script>
```

Then add the following line to master.blade.php where the javascript was previously:

```
@include('layouts.scripts')
```

Note we have to specify the folder, in this case "layouts." If the file you want is in the top level view folder, the welcome view for example, the syntax would be like this:

```
@include('welcome')
```

Obviously, we are not using that, it's just an example.

Using folders allows us to keep things more organized. And when we are following a RESTful format for the views, we will need to use index, create, edit, and show over and over, so those obviously require separate folders. We will be demonstrating this pattern in later chapters.

Anyway, if you followed the cut and paste instructions carefully and correctly, everything should still be working when you go to sample-project.com/test.

We're almost there. We can still do a little more organization on our master page. Let's start with the title. Chop it out and replace it with:

```
@yield('title')
```

By creating a yield injection point on the master page, we can define an @section('title') on the individual view pages, so we can have a custom title for each page.

Let's add a custom title to our test.index view. Put the following under the @extends, I will include that line for reference:

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

@section('title')
    <title>Test Page</title>

@endsection
```

So now when the composite page is rendered, the @section('title') will go into the master page at the point of @yield('title'). You can see how useful this is. We can easily create a custom title for every page.

For our template, we are going to create two more @yield injection points, one for css and for js.

Below the @include('layouts.css') line in master.blade.php, put the following line:

```
@yield('css')
```

And below the @include('layouts/scripts'), put the following line:

```
@yield('scripts')
```

So this allows you to bring in css and scripts into your master page from the individual views, just like we brought the title in from our test view, as long as you add a corresponding section to the view like we did for the title. Many times you will have unique requirements for a specific view and there's no reason to load extra js or css on every page if you don't have to.

You now have quite a bit of flexibility in how you can pull in the css and js that you need. Sometimes precedence is critical, this makes it very easy to see the order of things.

The benefits to Blade's organizational capabilities will become more clear as we build out our application.

Full Code

For reference and if you need to troubleshoot, I'm going to give you the complete layouts.master file as of this chapter via Gist:

[master.blade.php](#)

Here is css.blade.php via Gist:

[css.blade.php](#)

Here is scripts.blade.php via Gist:

[scripts.blade.php](#)

The scripts file only contains a single line at this point, but we are doing this for organizational purposes.

Stick with this format for now. We will do more refactoring in later chapters.

Summary

We covered a lot of ground quickly. We set up our connection to the DB and got introduced to the .env file, where we entered the host, username and password for the DB. We had a very brief introduction to unit testing. We don't do a lot of unit testing in this book, but at least you got to see that Laravel comes with it integrated and ready to work with.

We quickly moved on building our first route and controller named test, which will come in handy later when we need to work with some code and don't want to muck up an existing file.

We also made our first controller via Artisan, Laravel's command line tool. It was really easy, wasn't it? We will be using Artisan to make all kinds of files and the cool thing about that is Artisan knows which folders to put them in and adds the namespace for us, so we never trip over putting the new class in the wrong place.

Then we moved into working with views and got introduced to Blade, Laravel's amazing templating engine, which makes working with the front-end a pleasant experience.

Finally, we built our master page, giving us our first hint of what our sample-project is actually going to look like. Along the way we learned how to separate out our code into partials to give us maximum flexibility for working with it later on.

We haven't worked with models yet, but we will begin doing that in later chapters. Now I'm going to go commit my code and push it up to my repository. See you in the next chapter.

Chapter 5: User Registration And Login

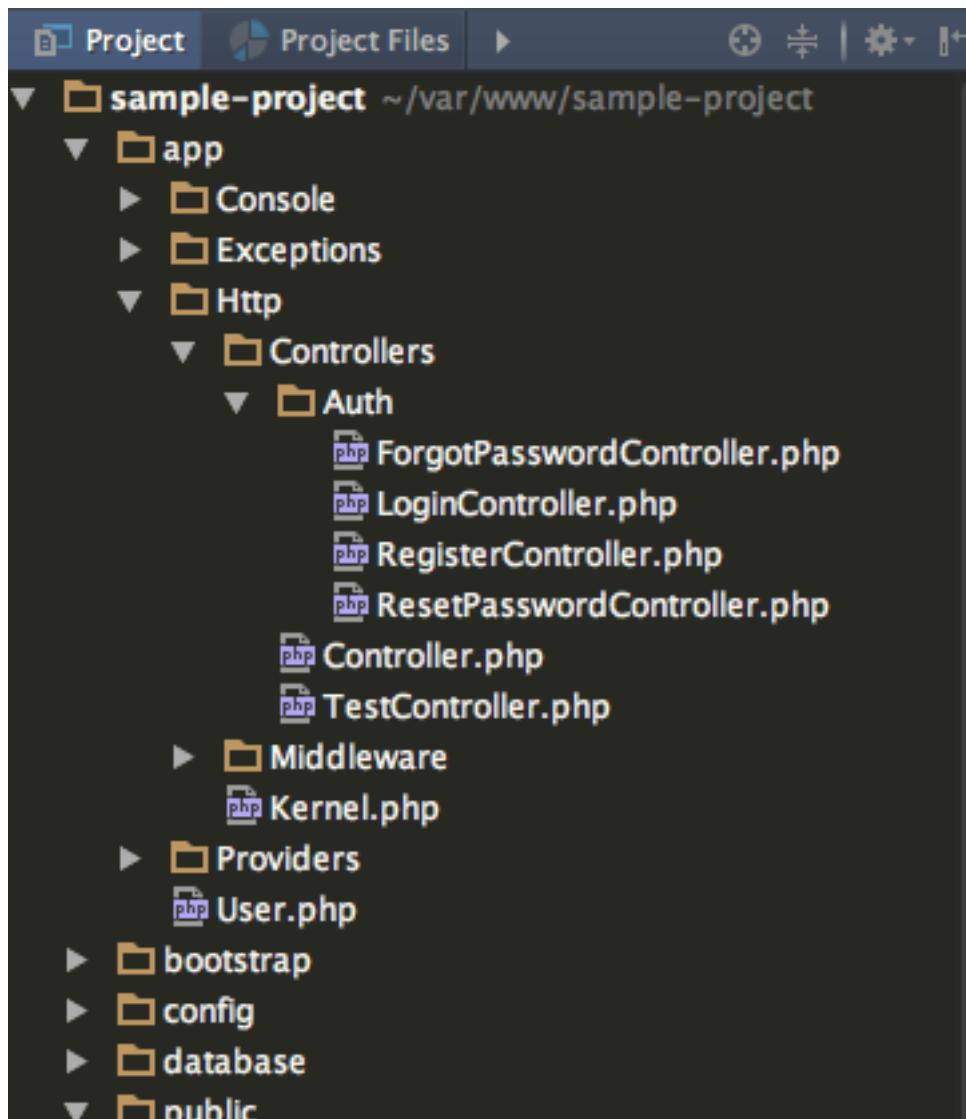
For most of our projects, we're going to need a user registration and login feature for our users. Building this can be fairly complicated. It involves setting up a user model, forgot password functionality, validation, etc.

Luckily for us, Laravel gets us most of the way there, right out of the box, with a simple artisan command. But before we run the command, let's understand the structure.

Controllers

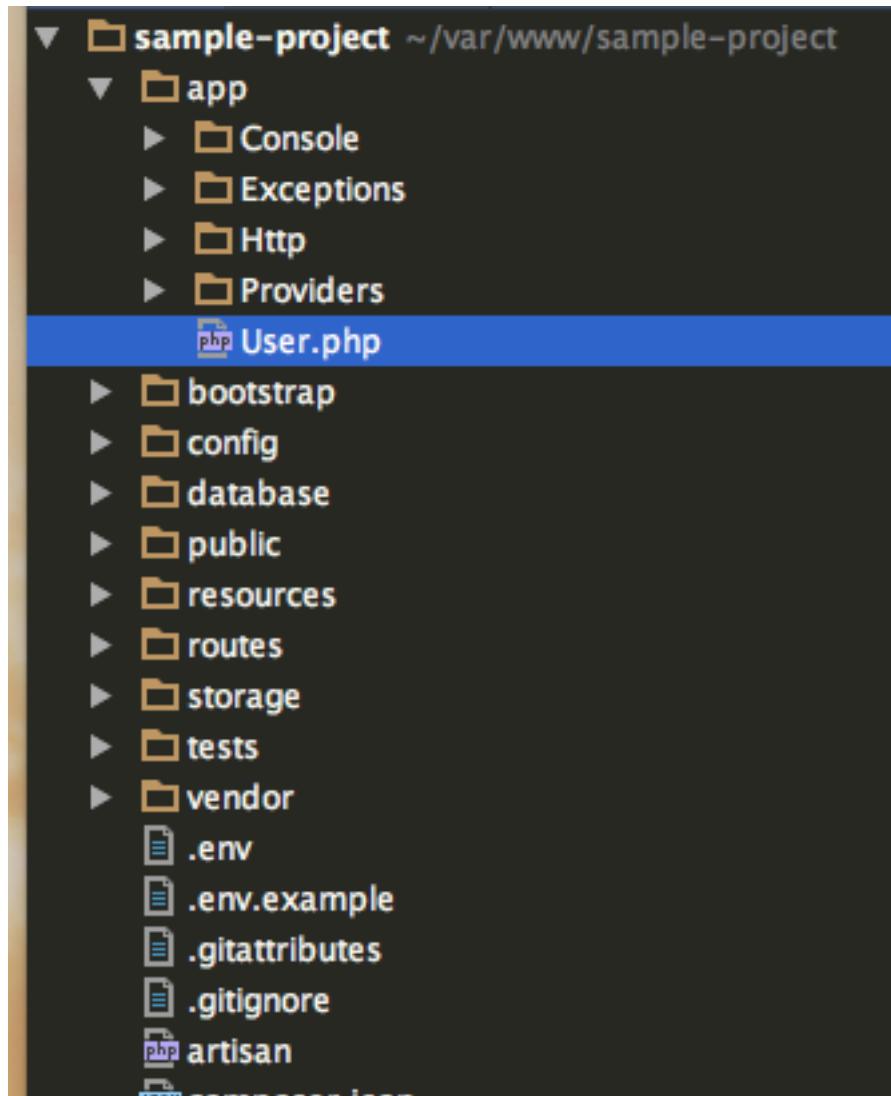
If you go to `app/Http/Controllers`, you will see an `Auth` folder, and within that are four controllers:

- `LoginController`
- `RegisterController`
- `ForgotPasswordController`
- `ResetPasswordController`:



User Model

Laravel also provides us with the user model:

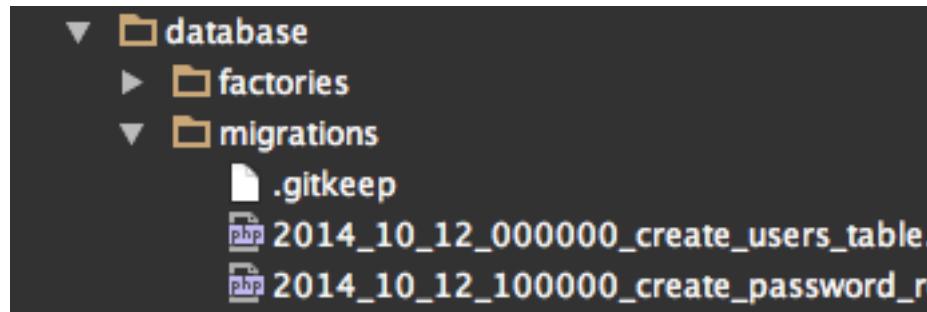


Models in Laravel don't get their own folder, they reside in the app folder. At first I thought that was wacky, very foreign to what I was used to. But once again in practical use, it just works out really well.

Migrations

We also get couple of migrations to setup the database tables. For anyone who might not be familiar with what a migration is, it is simply a set of instructions to the database from a php file. Let's take a look at the `create_users_table` migration that comes out of the box. You can find it in the migrations folder at the following path:

```
sample-project/database/migrations
```



So let's open the first one, the one for the users table:

```
<?php

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

class CreateUsersTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('users', function (Blueprint $table) {

            $table->increments('id');
            $table->string('name');
            $table->string('email')->unique();
            $table->string('password');
            $table->rememberToken();
            $table->timestamps();

        });
    }

    /**
     * Reverse the migrations.
     *

```

```
* @return void
*/
public function down()
{
    Schema::dropIfExists('users');
}
```

So there are two methods, up and down. The up method instructs the DB as to which changes you want. The down method is used for rollback purposes.

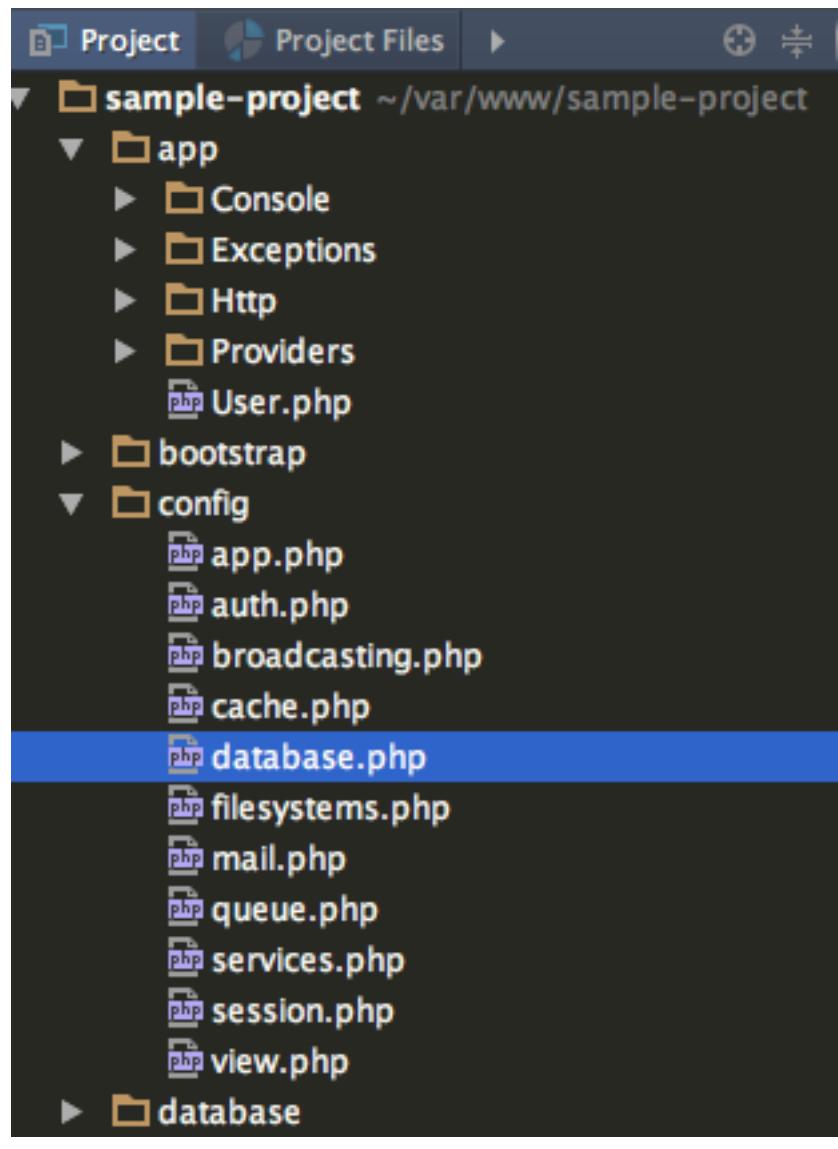
Note that when we create tables, we use a specific convention of plural. We have a users table, which will be accessed by a User model. The easy way to remember this is that the tables are meant to hold many instances, while the model returns a specific row. But obviously that's an oversimplification because of course many times we will be able to return more than one result. It's just a helpful way to remember the naming convention.

When I first learned about migrations, I didn't like them. I was convinced that you needed robust development tools like MySql workbench to properly build a data structure. However, after working with migrations, I find they are easier to work with, and it's easier to keep track of what you are doing as the migrations act as a version control to your database.

If you are keeping all of your work in a repository like Github, using migrations makes it easy to collaborate. So there are a lot of reasons why migrations are a good thing. Now I wouldn't dream of developing without them.

The [Laravel Docs](#) on migration commands are very clear and complete, so you can always find a reference for what you need.

Before we migrate, we are going to have to make one small change to sample-project/config/database.php, which is located here:



database.php

We're going to modify two of our 'mysql' settings:

```
'charset' => 'utf8mb4',
'collation' => 'utf8mb4_unicode_ci',
```

Change them to:

```
'charset' => 'utf8',
'collation' => 'utf8_unicode_ci',
```

We're doing this because older versions of MySQL may return errors in character length for the table when we do our migration. If you have a MySQL version below 5.7, this is a problem, and since I'm recommending MAMP, which as of this writing comes with 5.6.28, we are simply changing the setting, it's the easiest way to go.

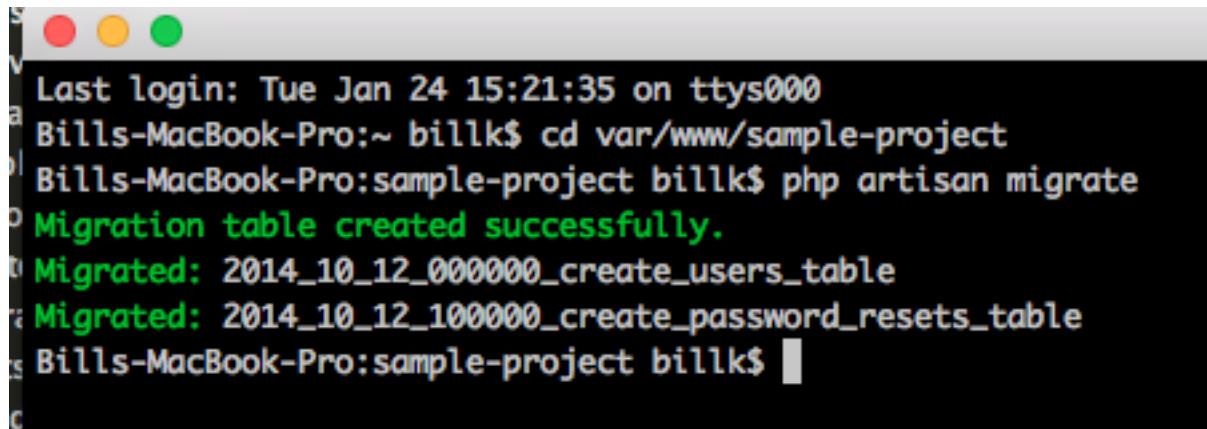
For our purposes, the gain in db optimization between the versions is not a factor, so we won't worry about that. Hopefully MAMP will be upgrading soon.

So now that we have that out of the way, we can demonstrate the migrations properly. As we mentioned before, we have two migrations sitting there that we are going to run.

Go to the command line and type the following:

```
php artisan migrate
```

You should get some instant feedback on the screen:



```
Last login: Tue Jan 24 15:21:35 on ttys000
Bills-MacBook-Pro:~ billlk$ cd var/www/sample-project
Bills-MacBook-Pro:sample-project billlk$ php artisan migrate
Migration table created successfully.
Migrated: 2014_10_12_000000_create_users_table
Migrated: 2014_10_12_100000_create_password_resets_table
Bills-MacBook-Pro:sample-project billlk$
```

Now to double-check it, let's look at our sample-project DB and see what we have. Go to Php MyAdmin, which you can reach through your MAMP start page:

	Table	Action							Rows	Type	Collation
<input type="checkbox"/>	migrations		Browse	Structure	Search	Insert	Empty	Drop	2	InnoDB	utf8_unicode_ci
<input type="checkbox"/>	password_resets		Browse	Structure	Search	Insert	Empty	Drop	0	InnoDB	utf8_unicode_ci
<input type="checkbox"/>	users		Browse	Structure	Search	Insert	Empty	Drop	0	InnoDB	utf8_unicode_ci
3 tables		Sum							2	InnoDB	utf8_unicode_ci

After 1st Migration

You can see we got three tables, one to track migrations, the other two are the expected tables, user and password_resets.

Ok, so now let's demonstrate rollback. Type the following in the command line:

```
php artisan migrate:rollback
```

And that will get you the following result:

	Table	Action							Rows	Type	Collation	Size
<input type="checkbox"/>	migrations		Browse	Structure	Search	Insert	Empty	Drop	0	InnoDB	utf8_unicode_ci	16 KiB
1 table		Sum							0	InnoDB	utf8_unicode_ci	16 KiB

[Print view](#) [Data Dictionary](#)

So yes, we lost the user and password_resets table, but we kept the migrations table. This is expected behavior, so all is well. We never modify the migrations table. Doing so can mess up your migration history and we don't want that, it can get ugly fast.

So now we just need to run the migration again and we will be back to what we want, which is to have our users and password_resets tables:

```
php artisan migrate
```

Make sure to double check that the tables have been created and we're good.

Laravel has a way to specify how many migrations you want to step back:

```
php artisan migrate:rollback --step=1
```

That can come in handy if you just want to rollback a specific migration. That's all we're going to do on migrations for now, we will be showing more implementations later in the book.

Make Auth

Laravel has this wonderful artisan command that will scaffold user authentication and password resets for us. It will create the routes we need as well as the views. The controllers are already in place. Once we run the command, we will step through all of this so you understand what has happened and how it works. You are going to love this.

```
php artisan make:auth
```

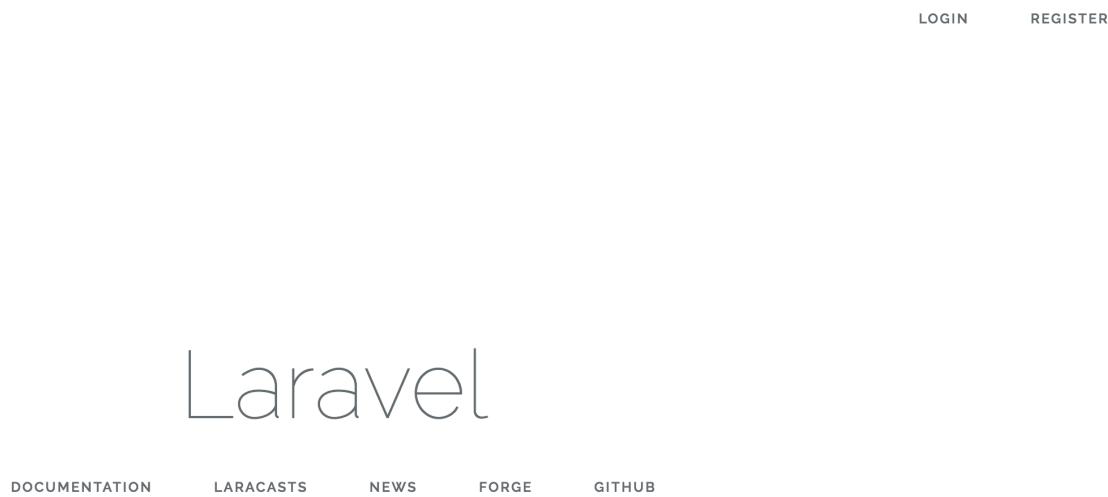
You will get the following response:

```
Bills-MacBook-Pro:sample-project billk$ php artisan make:auth
Authentication scaffolding generated successfully.
Bills-MacBook-Pro:sample-project billk$
```

It created a lot of view files for you and a controller, which we will see in a moment. Now if you visit:

www.sample-project

You will see the following:



So now you can hit the registration page and create a user.

The registration page looks like this:

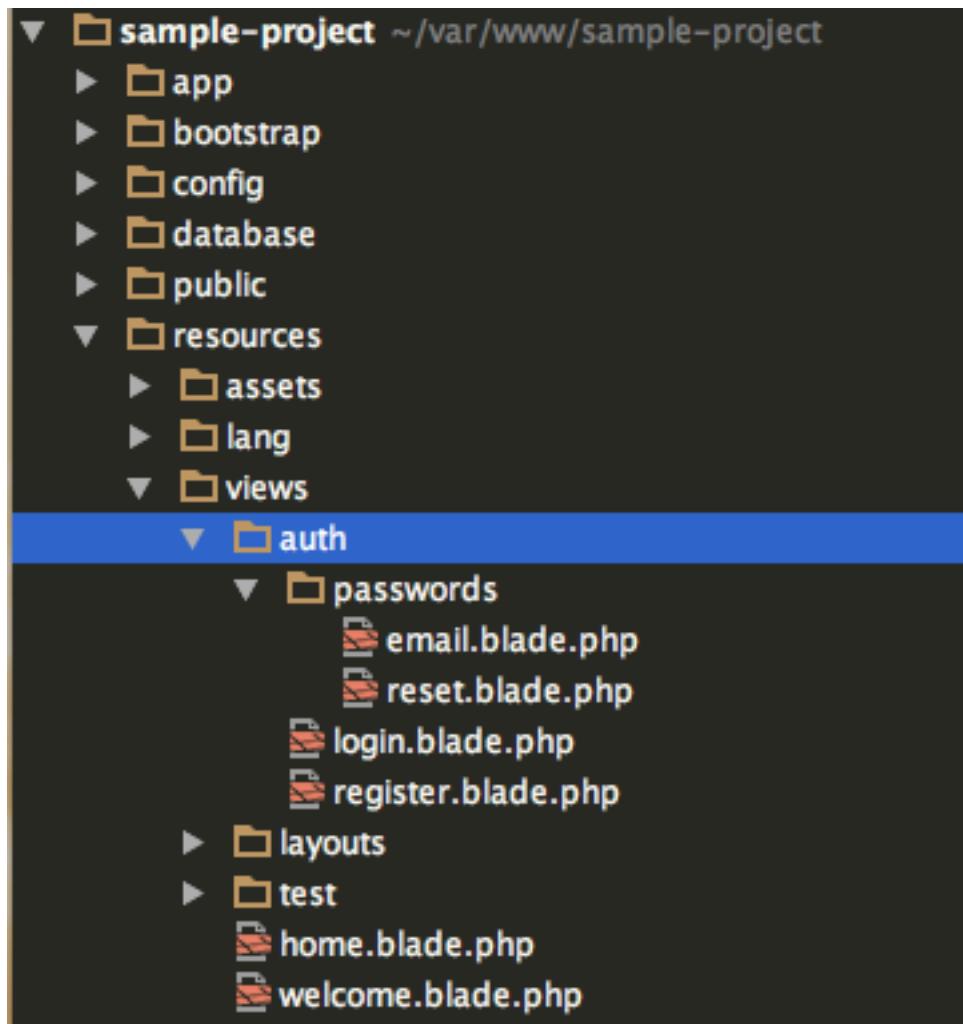
The screenshot shows a registration form titled "Register". The form has four input fields: "Name", "E-Mail Address", "Password", and "Confirm Password". Below the fields is a blue "Register" button. At the top right of the page, there are links for "Login" and "Register".

Go ahead and register now. You will see your username when you are logged in:

The screenshot shows a dashboard titled "Dashboard". A message "You are logged in!" is displayed. At the top right, there is a dropdown menu labeled "gumby ▾".

Ok, so you can log out from the drop down list, then use the login link to log back in so you can see it's working.

You can see all the blade files that were created in our new auth folder within views:



As you can see, not only did we get a full user registration and login set of views, we also got the password recovery views in the passwords folder as well.

Because we already had a layouts folder created, Laravel simply inserted app.blade.php into it. Otherwise it would have created that folder as well.

Since we are going to continue to build the application into a sample-project, we will not be using app.blade.php, once we have made the necessary changes in our other views to extend layouts.master instead of layouts.app, but we are not ready for that yet.

We're going to slow down a bit and take a few minutes to understand how the auth system works. It will be essential for you to know this when modifying it the future, as we will do, when we add socialite and modify the users table.

That said, don't worry about memorizing any of this, it's not necessary because all of this code is working out of the box and we don't need to anything. We're just going to get introduced to it.

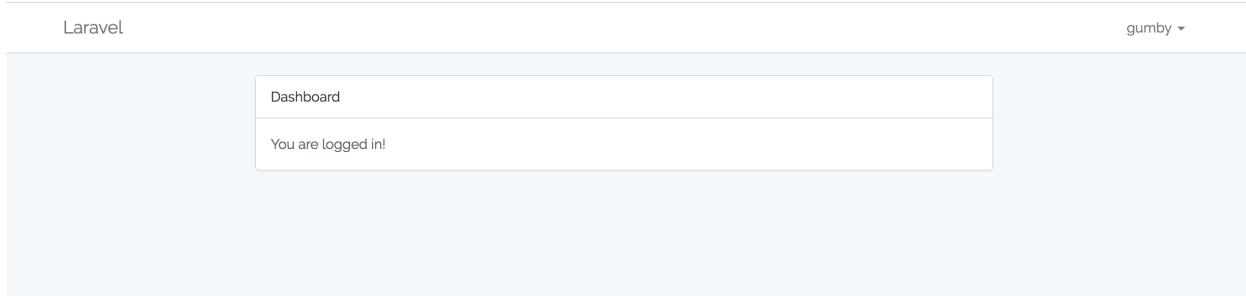
Ok let's jump in and start with the new route that was created. If you open your routes file, located at routes/web.php, you will see the following was added:

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

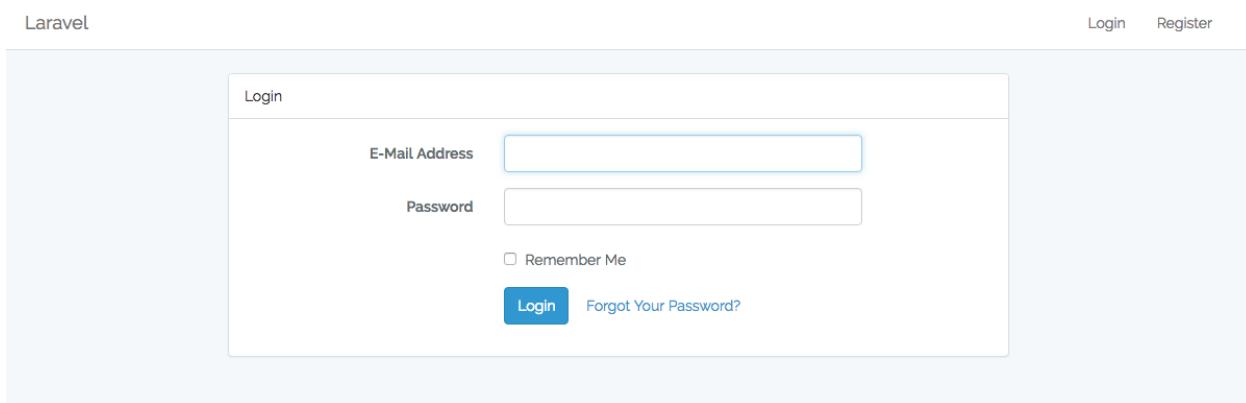
When you are logged in, if you type in the following url:

sample-project.com/home

You arrive here:



If you try the home uri in a logged out state, you can see it takes you to the login form:



Currently, the app redirects you '/home' upon login. That's because there is a property named `redirectTo` set on the `LoginController` that is set to '/home'. We will take a look at this, but before we do, let's understand how the auth and password routes work.

In the `web.php` route file, you have the following route:

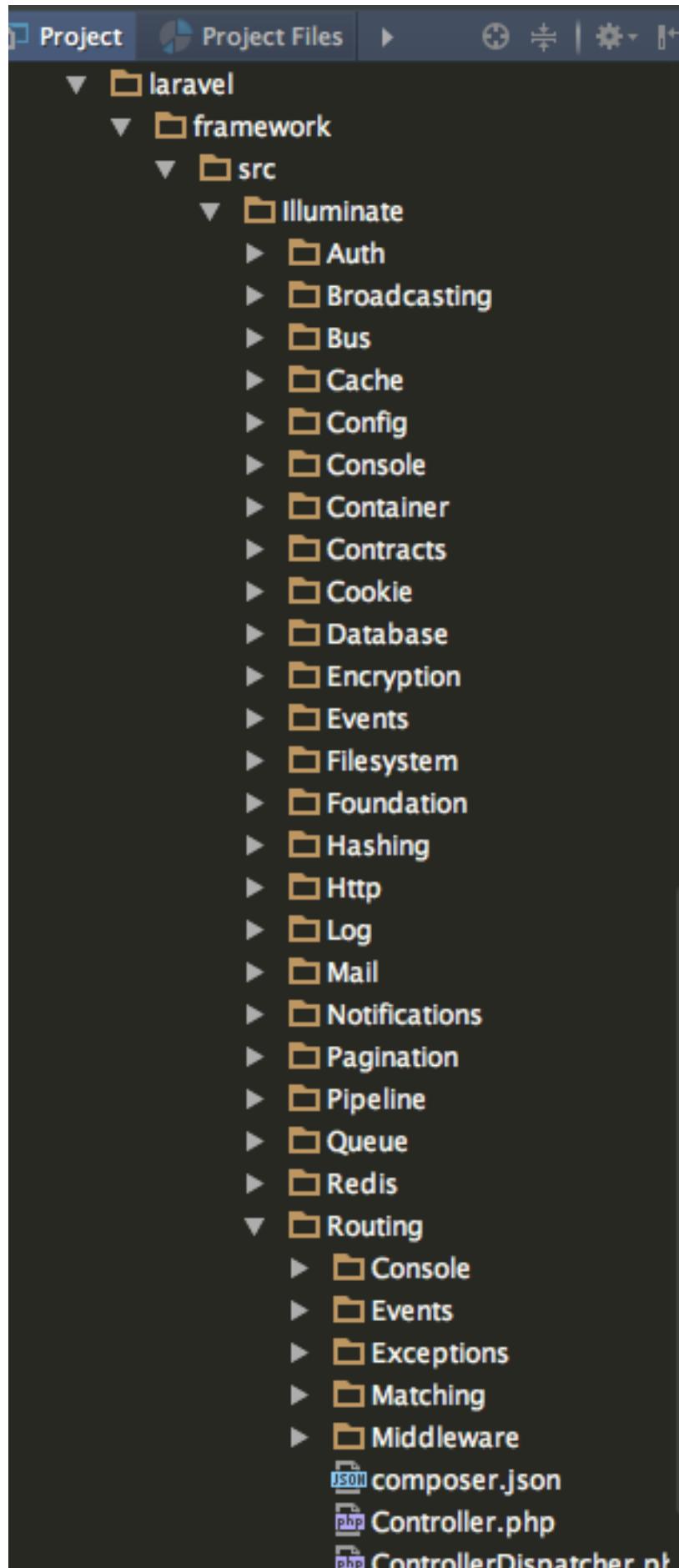
```
Auth::routes();
```

The routes method is in the Auth facade, which calls an instance of the Router class auth method with the following:

```
public static function routes()
{
    static::$app->make('router')->auth();
}
```

That tells the app to make an instance of the Router class's auth method. Through the magic of the framework, it knows exactly where the file is.

If we look deep in the framework, we can find the Router class that calls the auth method. It is located in the vendor directory:



It's in vendor/laravel/framework/src/Illuminate/Routing/Router.php. And inside that file, you find the auth method, which has the auth routes:

```
/*
 * Register the typical authentication routes for an application.
 *
 * @return void
 */
public function auth()
{
    // Authentication Routes...

    $this->get('login', 'Auth\LoginController@showLoginForm')->name('login');

    $this->post('login', 'Auth\LoginController@login');

    $this->post('logout', 'Auth\LoginController@logout')->name('logout');

    // Registration Routes...

    $this->get('register', 'Auth\RegisterController@showRegistrationForm')
        ->name('register');

    $this->post('register', 'Auth\RegisterController@register');

    // Password Reset Routes...

    $this->get('password/reset',
        'Auth\ForgotPasswordController@showLinkRequestForm')
        ->name('password.request');

    $this->post('password/email',
        'Auth\ForgotPasswordController@sendResetLinkEmail');

    $this->get('password/reset/{token}',
        'Auth\ResetPasswordController@showResetForm')->name('password.reset');

    $this->post('password/reset', 'Auth\ResetPasswordController@reset');

}
```

Note: I've added spaces for readability. Due to the limits of page formatting, what you see in the book will often differ, in terms of code style, from what you will see in your IDE. As I mentioned before, I'm doing this to make the code more clear and readable in the given format and to avoid line breaks that will otherwise break the code.

Anyway you can see that some of these routes are using the name method to name the route:

```
$this->get('login', 'Auth\LoginController@showLoginForm')->name('login');
```

Naming the route comes in handy when we have to refer to the route in other parts our code, typically in the controller. We will see an example of that later.

Let's look at another convention. If you look at the login routes for example, you can see that we need Auth\ in front of the controller, and that's because we have an Auth folder inside of the controllers folder. So when you are creating a folder inside of controllers, the convention for routing is:

NameOfFolder\ControllerName@method

Just a quick note. Obviously, we don't make changes to the vendor files because if we did, every time we do a composer update, they would get overwritten. So that means that if you want to build your own auth system or change the routing, you would not put your routes here. Instead they would go in the web.php file in your routes folder.

Ok, now that we know how the auth routes are being set, we can move on to the Auth controllers.

RegisterController

The RegisterController.php file is in the app/Http/Controllers/Auth folder. If you jumped ahead for whatever reason and looked in the RegisterController, you are probably asking yourself the same question I asked myself the first time I saw it, "Where the heck is everything?"

Where is the method to display the registration form? Where are the rest of the registration methods?

Well, they are extracted into a trait. Before diving into that, let's talk about traits for a minute.

Traits

Traits are relatively new to PHP, only available since PHP version 5.4, and they were new to me before I started working with Laravel. But I've come to appreciate them a lot.

If you would like to read up on traits, here is good article:

Traits

They describe it better than I can, but let me give it a try. Traits are a way of inheriting common methods. All you do is “use” the trait and then the traits methods become available to the class that is using it.

If you have a trait that has:

```
trait Calculates
{
    public function add($x, $y)
    {
        return $x + $y;
    }
}
```

We've all seen this kind of example method before, it just returns the sum of two numbers. Then, because it's a trait, you could use it with a completely different class, without extending the new class, by using it like so:

```
class CountFish
{
    use Calculates; // this pulls in the trait

    private $shelf = 3;
    private $bucket = 4;

    public static function totalFish()
    {
        //uses add method from trait

        $total = $this->add($this->shelf, $this->bucket);

        return $total;
    }
}

CountFish::totalFish();

// returns 7
```

So we get to use `$this` to call the `add` method, even though the `add` method doesn't reside in the class. This is a super simple and somewhat silly example, but I hope you get the idea.

Laravel has done a beautiful job of extracting out some of the methods from the `RegisterController` to a `RegistersUsers` trait, and all of those methods are available to the `RegisterController`.

Let's see if we can make sense of this by starting with the top level, the `RegisterController`:

```
<?php

namespace App\Http\Controllers\Auth;

use App\User;
use App\Http\Controllers\Controller;
use Illuminate\Support\Facades\Validator;
use Illuminate\Foundation\Auth\RegistersUsers;

class RegisterController extends Controller
{
    /*
    |--------------------------------------------------------------------------
    | Register Controller
    |--------------------------------------------------------------------------
    |
    | This controller handles the registration of new users as well as their
    | validation and creation. By default this controller uses a trait to
    | provide this functionality without requiring any additional code.
    |
    */
    use RegistersUsers;

    /**
     * Where to redirect users after registration.
     *
     * @var string
     */
    protected $redirectTo = '/home';

    /**
     * Create a new controller instance.
     *
     * @return void
     */
    public function __construct()
    {
        $this->middleware('guest');
    }

    /**
     * Get a validator for an incoming registration request.
     *

```

```

 * @param array $data
 * @return \Illuminate\Contracts\Validation\Validator
 */
protected function validator(array $data)
{
    return Validator::make($data, [
        'name' => 'required|max:255',
        'email' => 'required|email|max:255|unique:users',
        'password' => 'required|min:6|confirmed',
    ]);
}

/**
 * Create a new user instance after a valid registration.
 *
 * @param array $data
 * @return User
 */
protected function create(array $data)
{
    return User::create([
        'name' => $data['name'],
        'email' => $data['email'],
        'password' => bcrypt($data['password']),
    ]);
}
}

```

We get a nice comment telling us about the RegisterUsers trait.

Ok, something to important understand here. The RegisterController resides in your app/Http/Controllers/Auth folder. Running composer update will not change this file.

The RegistersUsers trait resides in your vendor/laravel/framework/src/Illuminate/Foundation/Auth folder. Running composer update may change the contents of this file if a new version of the framework includes changes to this file, which happens on a regular basis.

This is why we wouldn't typically make changes to the RegistersUsers trait, it will lead to headaches later on because it can be accidentally overwritten. For anyone who is just starting out with Laravel, I would highly

I recommend not changing the vendor framework code, unless you absolutely have to, and it's unlikely that would actually be necessary.

Ok, that said, let's look at the controller. You can see we are able to use the methods in RegistersUsers because we have a use statement in the class to make its methods available to us. Note that we are pulling in the trait with the use statement above the class:

```
use Illuminate\Foundation\Auth\RegistersUsers;
```

Then we have a protected property:

```
protected $redirectTo = '/home';
```

This sets the uri after login, which occurs after registration.

Next we have the constructor:

```
public function __construct()
{
    $this->middleware('guest');
}
```

We will be exploring the middleware concept in laravel in great detail, but for now, we'll simply say that this middleware is a filter that limits access to guests only. In other words, you can't register if you are already logged in, which makes a lot of sense.

Next you have two methods, validator and create:

```

/**
 * Get a validator for an incoming registration request.
 *
 * @param array $data
 * @return \Illuminate\Contracts\Validation\Validator
 */
protected function validator(array $data)
{
    return Validator::make($data, [
        'name' => 'required|max:255',
        'email' => 'required|email|max:255|unique:users',
        'password' => 'required|min:6|confirmed',
    ]);
}

/**
 * Create a new user instance after a valid registration.
 *
 * @param array $data
 * @return User
 */
protected function create(array $data)
{
    return User::create([
        'name' => $data['name'],
        'email' => $data['email'],
        'password' => bcrypt($data['password']),
    ]);
}

```

You can see the Validator class has a static make method to consume the request data and apply validation rules. We will be looking at validation in more detail later.

We also have a create method, which takes in the data and creates a user. But we know more than that happens. How does the user get logged in? For that matter, how does the registration form appear in the first place?

For that we need go to the RegistersUsers trait.

```
<?php

namespace Illuminate\Foundation\Auth;

use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;
use Illuminate\Auth\Events\Registered;

trait RegistersUsers
{
    use RedirectsUsers;

    /**
     * Show the application registration form.
     *
     * @return \Illuminate\Http\Response
     */
    public function showRegistrationForm()
    {
        return view('auth.register');
    }

    /**
     * Handle a registration request for the application.
     *
     * @param \Illuminate\Http\Request $request
     * @return \Illuminate\Http\Response
     */
    public function register(Request $request)
    {
        $this->validator($request->all())->validate();

        event(new Registered($user = $this->create($request->all())));

        $this->guard()->login($user);

        return $this->registered($request, $user) ?:
```

```
    redirect($this->redirectToPath());  
  
}  
  
/**  
 * Get the guard to be used during registration.  
 *  
 * @return \Illuminate\Contracts\Auth\StatefulGuard  
 */  
  
protected function guard()  
{  
    return Auth::guard();  
}  
  
/**  
 * The user has been registered.  
 *  
 * @param \Illuminate\Http\Request $request  
 * @param mixed $user  
 * @return mixed  
 */  
  
protected function registered(Request $request, $user)  
{  
    //  
}  
  
}
```

Right away, we answer the question of how the form appears:

```
public function showRegistrationForm()
{
    return view('auth.register');

}
```

The next method actually handles the registration:

```
public function register(Request $request)
{
    $this->validator($request->all())->validate();

    event(new Registered($user = $this->create($request->all())));

    $this->guard()->login($user);

    return $this->registered($request, $user) ?:
        redirect($this->redirectTo());
}
```

So what's happening is that the request is being run through the validator method, then it fires a new Registered event to create the user record. \$this->guard() method is then called, which sets Auth::guard(), and then we fluently chain the login method.

Note that the \$user variable is assigned to the create method:

```
event(new Registered($user = $this->create($request->all())));
```

So then we have access to the user for the login and registered methods.

Then it returns to the redirect path via the ternary:

```
return $this->registered($request, $user) ?:  
    redirect($this->redirectPath());
```

In this case, the registered method being empty is going to return null and evaluate false and force the redirect method.

Hey, where's that redirectPath method? It's in the RedirectsUsers trait, which is called in by the RegistersUsers trait, a nice example of nested traits. Here is the RedirectsUsers trait:

```
<?php  
  
namespace Illuminate\Foundation\Auth;  
  
trait RedirectsUsers  
{  
    /**  
     * Get the post register / login redirect path.  
     *  
     * @return string  
     */  
    public function redirectPath()  
    {  
        if (method_exists($this, 'redirectTo')) {  
  
            return $this->redirectTo();  
        }  
  
        return property_exists($this, 'redirectTo') ?  
            $this->redirectTo : '/home';  
    }  
}
```

We're checking to see if the redirectTo property is set and if so, use it. If not, default to '/home'.

And that's as deep as we need to go for now into registration, but at least we have a solid idea on how it works.

We will be looking at the routes and views, which give us the forms in the browser for both registration and login in a few minutes. But first let's move on to the LoginController.

```
<?php

namespace App\Http\Controllers\Auth;

use App\Http\Controllers\Controller;
use Illuminate\Foundation\Auth\AuthenticatesUsers;

class LoginController extends Controller
{
    /*
    |--------------------------------------------------------------------------
    | Login Controller
    |--------------------------------------------------------------------------
    |
    | This controller handles authenticating users for the application and
    | redirecting them to your home screen. The controller uses a trait
    | to conveniently provide its functionality to your applications.
    |
    */
    use AuthenticatesUsers;

    /**
     * Where to redirect users after login.
     *
     * @var string
     */

    protected $redirectTo = '/home';

    /**
     * Create a new controller instance.
     *
     * @return void
     */

    public function __construct()
    {
        $this->middleware('guest', ['except' => 'logout']);
    }
}
```

```
 }  
}
```

Once again, it's clean and simple. This time in the constructor, the middleware is applied to all methods except logout. Of course you don't see a logout method, do you?

For that we need to look at the AuthenticatesUsers trait:

```
<?php  
  
namespace Illuminate\Foundation\Auth;  
  
use Illuminate\Http\Request;  
use Illuminate\Support\Facades\Auth;  
  
trait AuthenticatesUsers  
{  
    use RedirectsUsers, ThrottlesLogins;  
  
    /**  
     * Show the application's login form.  
     *  
     * @return \Illuminate\Http\Response  
     */  
  
    public function showLoginForm()  
    {  
        return view('auth.login');  
    }  
  
    /**  
     * Handle a login request to the application.  
     *  
     * @param \Illuminate\Http\Request $request  
     * @return \Illuminate\Http\RedirectResponse|\Illuminate\Http\Response  
    }
```

```
*/  
  
public function login(Request $request)  
{  
  
    $this->validateLogin($request);  
  
    // If the class is using the ThrottlesLogins trait,  
    // we can automatically throttle  
    // the login attempts for this application. We'll key this  
    // by the username and  
    // the IP address of the client making these requests  
    // into this application.  
  
    if ($this->hasTooManyLoginAttempts($request)) {  
  
        $this->fireLockoutEvent($request);  
  
        return $this->sendLockoutResponse($request);  
    }  
  
    if ($this->attemptLogin($request)) {  
  
        return $this->sendLoginResponse($request);  
    }  
  
    // If the login attempt was unsuccessful we will  
    // increment the number of attempts  
    // to login and redirect the user back to the login  
    // form. Of course, when this  
    // user surpasses their maximum number of attempts  
    // they will get locked out.  
  
    $this->incrementLoginAttempts($request);  
  
    return $this->sendFailedLoginResponse($request);  
}  
  
/**  
 * Validate the user login request.  
 */
```

```
* @param  \Illuminate\Http\Request  $request
* @return void
*/
protected function validateLogin(Request $request)
{
    $this->validate($request, [
        $this->username() => 'required', 'password' => 'required',
    ]);
}

/**
 * Attempt to log the user into the application.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return bool
 */
protected function attemptLogin(Request $request)
{
    return $this->guard()->attempt(
        $this->credentials($request),
        $request->has('remember')
    );
}

/**
 * Get the needed authorization credentials from the request.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return array
 */
protected function credentials(Request $request)
{
    return $request->only($this->username(), 'password');
```

```
}

/**
 * Send the response after the user was authenticated.
 *
 * @param \Illuminate\Http\Request $request
 * @return \Illuminate\Http\Response
 */

protected function sendLoginResponse(Request $request)
{
    $request->session()->regenerate();

    $this->clearLoginAttempts($request);

    return $this->authenticated($request,
        $this->guard()->user()) ?:
        redirect()->intended($this->redirectPath());

}

/**
 * The user has been authenticated.
 *
 * @param \Illuminate\Http\Request $request
 * @param mixed $user
 * @return mixed
 */

protected function authenticated(Request $request, $user)
{
    //
}

/**
 * Get the failed login response instance.
 *
 * @param \Illuminate\Http\Request $request
 * @return \Illuminate\Http\RedirectResponse
 */

protected function sendFailedLoginResponse(Request $request)
{
```

```
$errors = [$this->username() => trans('auth.failed')];

if ($request->expectsJson()) {

    return response()->json($errors, 422);

}

return redirect()
    ->back()
    ->withInput($request
        ->only($this->username(), 'remember'))
    ->withErrors($errors);
}

/** 
 * Get the login username to be used by the controller.
 *
 * @return string
 */

public function username()
{
    return 'email';
}

/** 
 * Log the user out of the application.
 *
 * @param \Illuminate\Http\Request $request
 * @return \Illuminate\Http\Response
 */

public function logout(Request $request)
{
    $this->guard()->logout();

    $request->session()->flush();

    $request->session()->regenerate();

    return redirect('/');
}
```

```
}

/**
 * Get the guard to be used during authentication.
 *
 * @return \Illuminate\Contracts\Auth\StatefulGuard
 */

protected function guard()
{
    return Auth::guard();
}

}
```

This trait actually uses two others:

```
use RedirectsUsers, ThrottlesLogins;
```

We've already seen `RedirectsUsers`, so we know what that does. `ThrottlesLogins` is a guard against excessive login attempts, which is a type of attack to not only gain unauthorized entry to the application, but also degrades application performance with too many requests.

So that's a nice feature Laravel is giving you right out of the box to protect against that. When you have time, check out the trait, in the interests of time and not overwhelming you with too much information, I'm not going to cover it in detail here.

Let's go back to the `AuthenticatesUsers` trait, which gets us the login form:

```
public function showLoginForm()
{
    return view('auth.login');
```

Next let's look at the login method itself:

```
public function login(Request $request)
{
    $this->validateLogin($request);

    // If the class is using the ThrottlesLogins trait,
    // we can automatically throttle
    // the login attempts for this application. We'll key this
    // by the username and
    // the IP address of the client making these requests
    // into this application.

    if ($this->hasTooManyLoginAttempts($request)) {

        $this->fireLockoutEvent($request);

        return $this->sendLockoutResponse($request);
    }

    if ($this->attemptLogin($request)) {

        return $this->sendLoginResponse($request);
    }

    // If the login attempt was unsuccessful we will
    // increment the number of attempts
    // to login and redirect the user back to the login
    // form. Of course, when this
    // user surpasses their maximum number of attempts
    // they will get locked out.

    $this->incrementLoginAttempts($request);

    return $this->sendFailedLoginResponse($request);
}
```

The login method takes the request and runs it through the validateLogin method:

```
protected function validateLogin(Request $request)
{
    $this->validate($request, [
        $this->username() => 'required', 'password' => 'required',
    ]);
}
```

Login cannot proceed unless it validates the username and password. `$this->username` is set to ‘email’. If for example, you wanted login to compare ‘name’ on the users table instead of email, you would overwrite the `username` method on the `LoginController`, not change the trait for the reasons I stated before, which is you don’t want to change vendor source code in most cases.

So the login method handles the validation, checks to see if any the lockout conditions apply from login throttling, then if all is good there, uses the attemptLogin method to login:

```
if ($this->attemptLogin($request)) {

    return $this->sendLoginResponse($request);
}
```

Here is the attemptLogin method:

```
protected function attemptLogin(Request $request)
{
    return $this->guard()->attempt(
        $this->credentials($request), $request->has('remember')
    );
}
```

`$this->guard()` returns `Auth::guard()`, which calls to the `StatefulGuard` interface, in `vendor/framework/src/Illuminate/Contracts/Auth/Access/StatefulGuard.php`, which extends the `Guard` interface in the same folder. I'm mentioning that in case you are curious where the `Auth` methods are located.

To finish up login, if the attempt is successful, then it calls the `sendLoginResponse` method:

```
protected function sendLoginResponse(Request $request)
{
    $request->session()->regenerate();

    $this->clearLoginAttempts($request);

    return $this->authenticated($request,
        $this->guard()->user()) ??
        redirect()->intended($this->redirectPath());
}
```

We also have a logout method:

```
public function logout(Request $request)
{
    $this->guard()->logout();

    $request->session()->flush();

    $request->session()->regenerate();

    return redirect('/');
}
```

You can see it's logging the user out, flushing the session, regenerating a new session, and finally redirecting to '/'.

I'm not going to go over every method on this trait, it's just too much, we would be going too deep. You can explore it all on your own.

The last point here is to move back to the LoginController and set the redirectTo property:

```
protected $redirectTo = '/home';
```

That is where you indicate where you want to land after login.

So this was quick tour through the RegisterController and LoginController, giving us a robust example of using traits. It's a little more intermediate than beginner, but I felt it was important to at least introduce you to it.

The good news is that most of our subsequent work with traits is going to be a lot easier to understand. So just hang in there if you don't quite get it.

As it stands now, the traits we need are supplied to us and work as is, so we really don't need to do anything with them.

The ResetsPasswordContorller and PasswordController use a similar techniques with their related traits. The top level controller is extremely light and the methods are extracted out to traits.

Like I mentioned before, this is a very sensible and flexible way to divide the methods, so if you work on your controller with a lot of custom methods, you are not interfering with the methods on the trait, unless you intentionally want to overwrite them in the controller. We will show an example of that later in the book.

Instead of going over each method in the password controllers, which you can explore on your own, let's work on one piece of configuration. If you try to use the password recovery link, it will ask you to supply an email address, so it can send you the reset link. However, if you try this now, you will get:

```
Swift_TransportException in AbstractSmtpTransport.php line 383:Expected response code 250 but got code "530", with message "530 5.7.1 Authentication required"
```

The reason this occurred is that we did not configure our email in our .env file.

Out of the box, the mail driver part of .env comes like this:

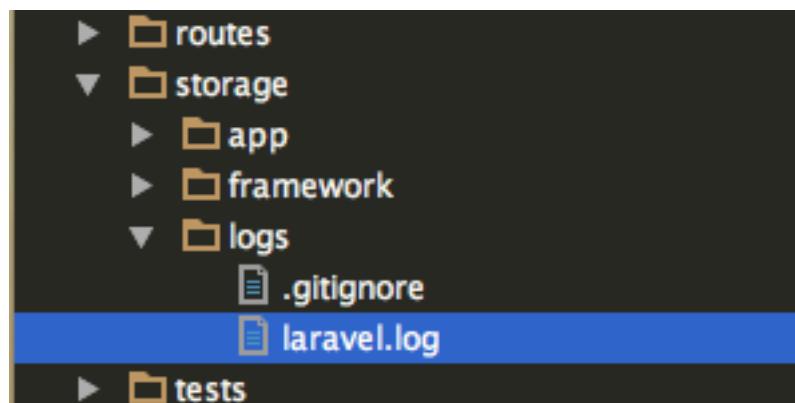
```
MAIL_DRIVER="smtp"
MAIL_HOST="mailtrap.io"
MAIL_PORT=2525
MAIL_USERNAME=null
MAIL_PASSWORD=null
MAIL_ENCRYPTION=null
```

It is setup for the mailtrap.io service, which is free and extremely useful for developers to use. It sends any emails from your application to your account, so you can build and test and see the results. Since we are not giving it our username and password for our mailtrap.io account, we get the error described above.

Before we set up mailtrap.io, let me show you another configuration that is simpler. Just change to the following:

```
MAIL_DRIVER=log
```

Now if you use the forgot password link form to send the reset email, you will find a copy of that email in:



If you look in that file, you can see you are getting the email, and you can grab the reset url out of the body to finish resetting your password.

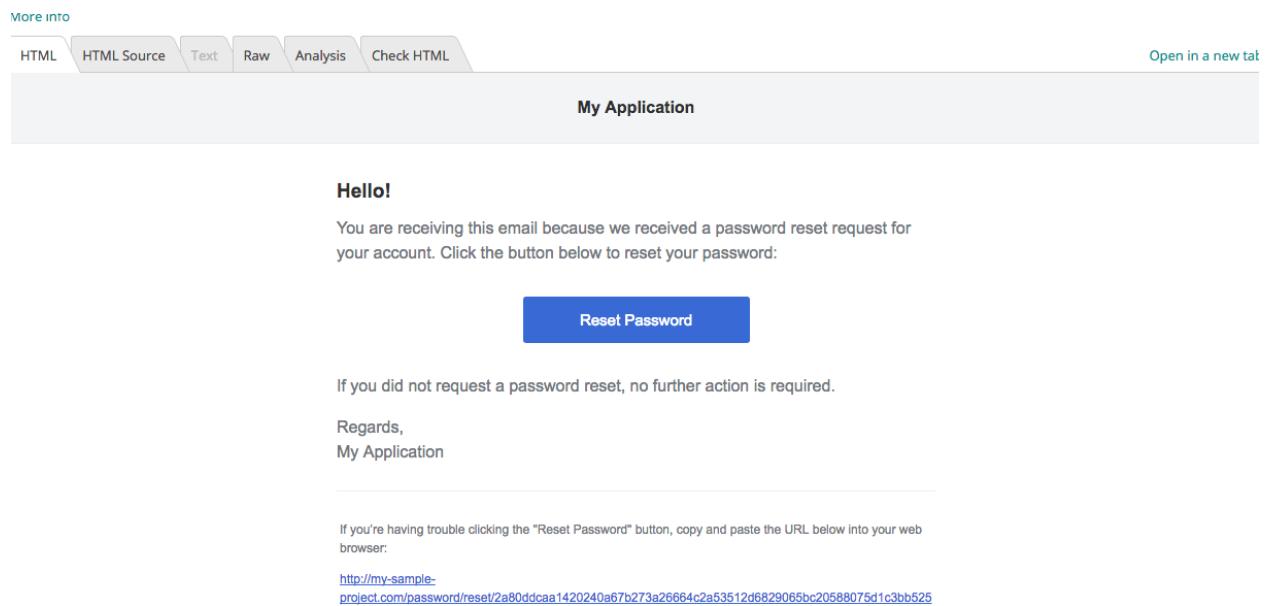
An even better way to do this, so you can view the html email, is to set up a mailtrap.io account, then supply your username and password in the .env file:

```
MAIL_DRIVER="smtp"
MAIL_HOST="mailtrap.io"
MAIL_PORT=2525
MAIL_USERNAME=your-username
MAIL_PASSWORD=your-password
MAIL_ENCRYPTION=null
```

Obviously, you will be substituting your actual username and password into the .env, not using the above.

I used my Github account for a one-click registration to mailtrap.io. Then click on the Demo inbox and grab your username and password, plug those into your .env and you're done.

Then try the forgot password form again, then check your mailtrap.io account, you will get an email that looks like this:



The screenshot shows an email from 'My Application' with the subject 'Hello!'. The email content includes a greeting, a password reset link, and instructions for users who did not request it. It also provides a URL for manual entry if needed.

More info [HTML](#) [HTML Source](#) [Text](#) [Raw](#) [Analysis](#) [Check HTML](#) [Open in a new tab](#)

My Application

Hello!

You are receiving this email because we received a password reset request for your account. Click the button below to reset your password:

[Reset Password](#)

If you did not request a password reset, no further action is required.

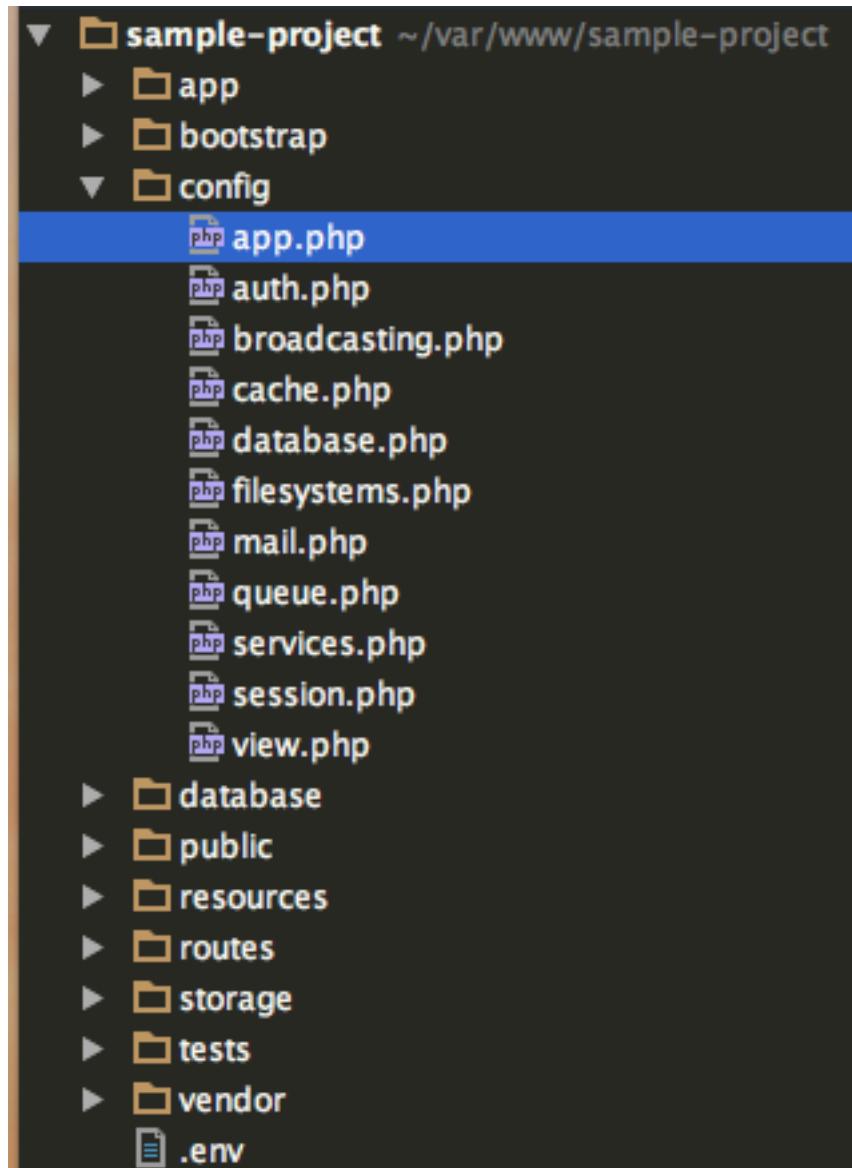
Regards,
My Application

If you're having trouble clicking the "Reset Password" button, copy and paste the URL below into your web browser:

<http://my-sample-project.com/password/reset/2a80ddcaa1420240a67b273a26664c2a53512d6829065bc20588075d1c3bb525>

Now that we've tested our forgot password functionality two ways, we can take a second to appreciate how cool it is that this is all provided for out of the box.

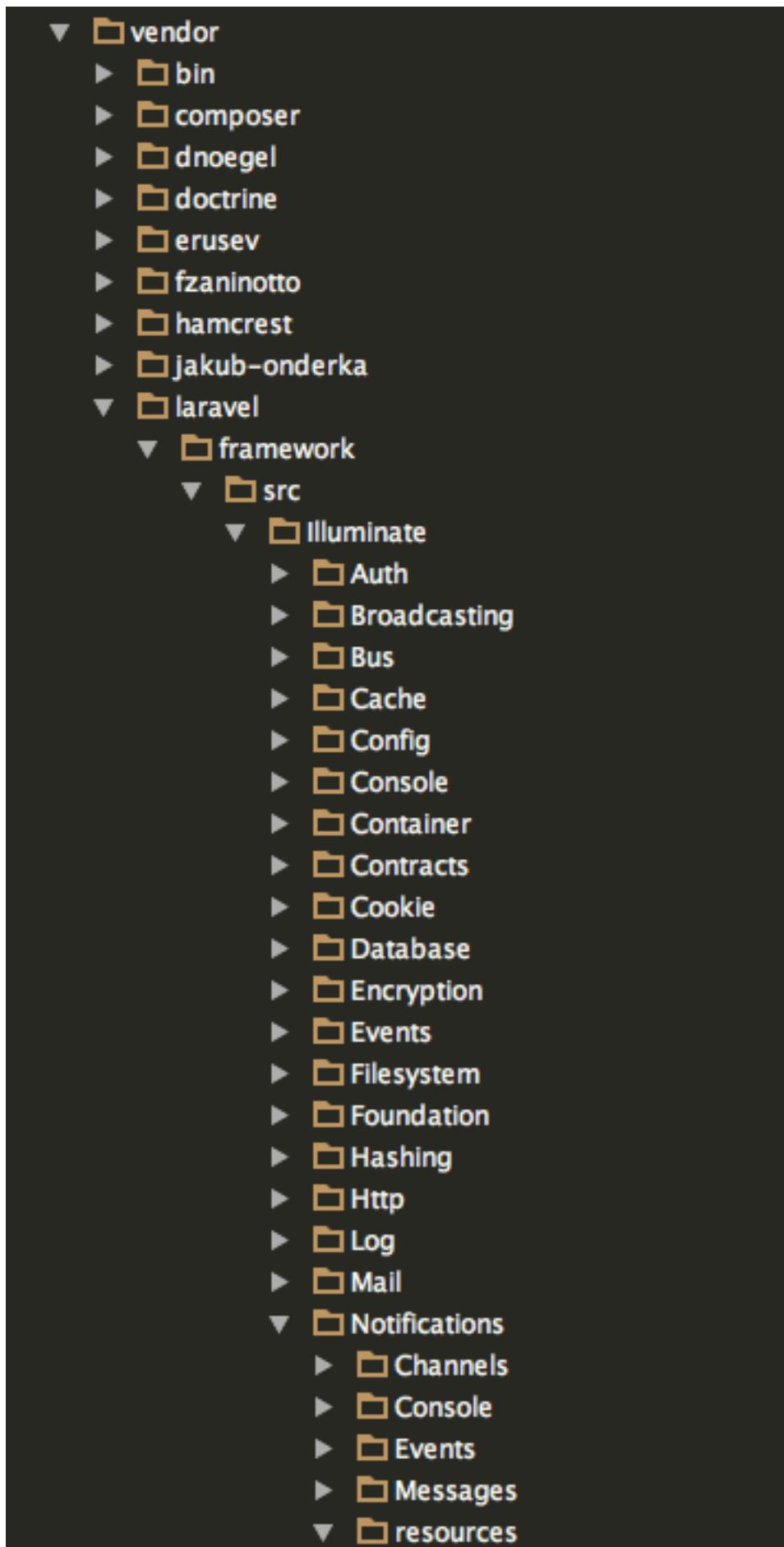
The top line that says “My Application” will actually say “Laravel” out of the box. If you want to change this, go to sample-project/config/app.php:



In the name key, you can see it's set to laravel, let's change this:

```
'name' => 'sample-project',
```

While the prefab email is cool, you might want to customize it. Right now the notification email lives in vendor/laravel/framework/src/Illuminate/Notifications/resources/views:

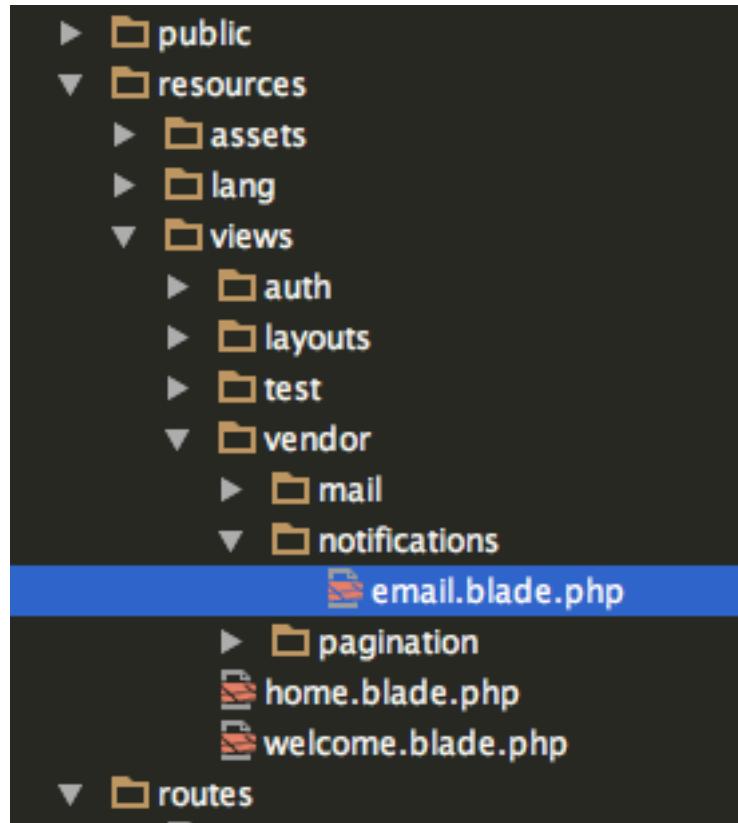


That's not really manageable for us, so we are going to publish the vendor assets, which will create a copy of it in our sample-project/resources/views folder.

Go to the command line and type the following command:

```
php artisan vendor:publish
```

This will create a vendor folder in our views folder, with three new folders within it, mail, notification, and pagination:



So now if you change email.blade.php, it will be reflected in the reset password link email. This template is obviously not appropriate for all scenarios, so we will look at sending a custom email in a later chapter in the book.

Ok, so our login, registration, and forgot password functionality is all in place, but the overall presentation of our application is a bit of a mess. We have to do some clean up.

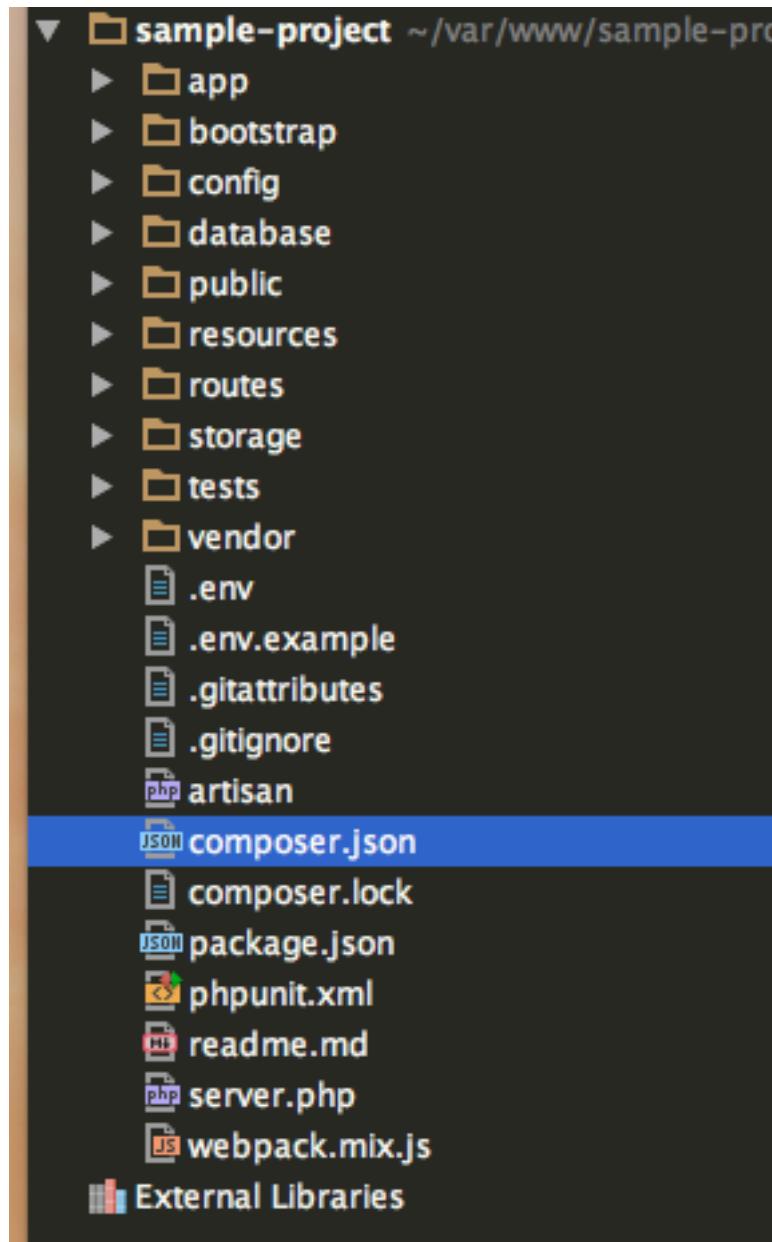
We're almost ready to walk through our view files, but before we do that, we are going to do some more work in our layouts.master view file. This is because all the views that were created for us by Laravel extend layouts.app and we want them to use layouts.master.

One cool feature I would have in most of my sites is a gravatar, which will display an image of the user's choosing upon login. And there just happens to be a handy little package to help us implement that, so we're going to pull in the package before going further.

Gravatar

Ok, so let's pull in the package by [CreativeOrange](#). You should check the installation instructions there, nevertheless I will provide them. Normally, we don't run composer update to add packages to our project. For beginners, I'm going to do it this time, so you see how composer update works. For all other packages, we will use composer require instead, and feel free to do it that way if you already know how.

We need to open composer.json, located here:



Via: composer update

In the require section of your composer.json, add the following line:

```
"creativeorange/gravatar": "~1.0"
```

Then from the command line run:

```
composer update
```

Next we go to sample-project/config/app.php and add the following to the providers array. There's a section that's commented for packages that already includes Tinker. Let's put our provider there:

```
Creativeorange\Gravatar\GravatarServiceProvider::class,
```

And in the aliases array:

```
'Gravatar' => Creativeorange\Gravatar\Facades\Gravatar::class,
```

A quick note on service providers. Think of them as the building blocks of your application. One of the really beautiful aspects of Laravel's architecture is the use of service providers to build and extend the application.

Facades

So logically, looking at the string above, you may ask, what is a facade? The first sentence in the [laravel docs on Facades](#):

“Facades provide a “static” interface to classes that are available in the application’s service container.”

We’ve already seen calls in laravel like Route::get, which you would normally think of as a static method, but Laravel innovates its way into providing the facade as a proxy, so the method used is not necessarily static. And of course we saw the Auth facade being used in the Auth traits.

The net effect and less technical side of it is that you get wonderfully expressive syntax without sacrificing testability.

Ok, perfect. Next we need to make an addition to our css.blade.php file, which is in views/layouts folder. Add the following to the file:

```
.circ{  
  
    width : 40px;  
    vertical-align: middle;  
    border-radius: 50%;  
    box-shadow: 0 0 0 2px #fff;  
    margin-top: 5px;  
  
}  
.circ:hover{  
  
    box-shadow: 0 0 0 3px #f00;  
  
}
```

This will allow us to draw a circle around the gravatar. We will use this once we change our layouts.master file.

Later in the book, we are going to work with Laravel Mix, so this is just a temporary home for our css, but we are going to keep it extremely simple for now.

Pages Controller

One more thing to do before we move onto layouts.master. The layouts.master file is going to be extended from our index page, but we don't have an index page yet. We are going to replace the welcome view and the HomeController with a controller for our index page.

We're going to create a PagesController, whose purpose will be to control pages like index, about, contact, and any other simple pages of the site.

If we follow the workflow that we established earlier in the book, we will make a route, a controller, a view folder, and a view, in that order.

So let's open web.php, which if you recall, is in the routes folder. Delete the existing home route:

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

Then add the new one like so:

```
// home page route  
  
Route::get('/', 'PagesController@index');
```

Ok, great. Do you remember how to create a controller using Artisan?

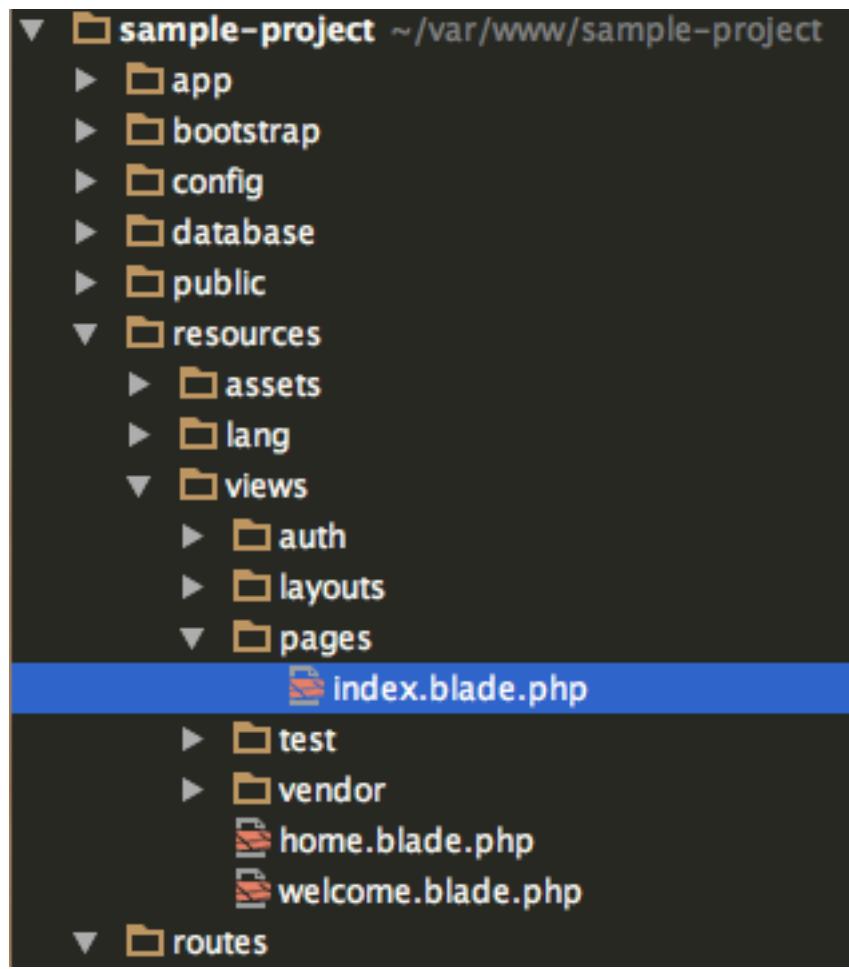
```
php artisan make:controller PagesController
```

Now let's add the index method on the PagesController.php file:

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

Ok, now we need a folder for our pages views, so let's make one and name it pages.

Next we need our index.blade.php file, so create that in the pages folder:



pages/index

Place the following in the index.blade.php file.

Gist:

[index.blade.php](#)

From book:

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

@section('title')

<title>Sample Project</title>

@endsection

@section('content')

<ol class="breadcrumb">
    <li><a href="#">Home</a></li>
</ol>

<!-- Main jumbotron for a primary marketing message or call to action --&gt;

&lt;div class="jumbotron"&gt;

    &lt;h1&gt;Sample Project&lt;/h1&gt;

    &lt;p&gt;Use it <b>as a starting point to create something more unique by
        building on or modifying it.
    </p>

</div>

@endsection
```

I grabbed the jumbotron from:

<http://getbootstrap.com/examples/theme/>

Anything you want to get from that sample page, you can just view source and grab it.

New Nav

Ok, so now we're finally ready for our new master.blade.php and we have something cool to show you with that. Let's start by changing master.blade.php to the following.

Gist:

[master.blade.php](#)

From book:

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

    <!-- The above 3 meta tags *must* come first in the head;
        any other head content must come *after* these tags -->

    <meta name="description"
          content="">
    <meta name="author"
          content="">

    <link rel="icon" href="../../favicon.ico">

    @yield('title')

    @include('layouts.css')

    @yield('css')

</head>

<body role="document">

    <!-- Fixed navbar -->

    <nav class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">

                <button type="button"
                       class="navbar-toggle collapsed"
                       data-toggle="collapse"
                       data-target="#navbar" aria-expanded="false"
                       aria-controls="navbar">
```

```
<span class="sr-only">Toggle navigation</span>
<span class="icon-bar"></span>
<span class="icon-bar"></span>
<span class="icon-bar"></span>

</button>

<a class="navbar-brand"
   href="/">Sample Project</a>
</div>

<div id="navbar"
      class="navbar-collapse collapse pull-right">

<ul class="nav navbar-nav">

<li class="active">
    <a href="/">
        Home
    </a>
</li>
<li>
    <a href="#about">
        About
    </a>
</li>
<li class="dropdown">
    <a href="#">
        class="dropdown-toggle"
        data-toggle="dropdown"
        role="button"
        aria-haspopup="true"
        aria-expanded="false">
<span class="caret"></span></a>
<ul class="dropdown-menu">
```

```
<li>
  <a href="/widget">
    Widgets
  </a>
</li>
</ul>

</li>

@if (Auth::check())

<li class="dropdown">
  <a href="#" class="dropdown-toggle"
      data-toggle="dropdown"
      role="button"
      aria-haspopup="true"
      aria-expanded="false">
    {{ Auth::user()->name }}
    <span class="caret"></span>
  </a>

  <ul class="dropdown-menu">
    <li>
      <a href="/logout"
          onclick="event.preventDefault();
                    document.getElementById('logout-form').submit();">
        Logout
      </a>
    </li>
  </ul>
</li>
```

```
    {{ csrf_field() }}
```

```
</form>
```

```
</li>
```

```
</ul>
```

```
</li>
```

```
<li>
</li>
```

```
@else
```

```
<li><a href="/login">
    Login
</a>
</li>
```

```
<li><a href="/register">
    Register
</a>
</li>
```

```
@endif
```

```
</ul>
```

```
</div><!--/.nav-collapse -->
```

```
</div>
```

```
</nav>
```

```
<div class="container theme-showcase" role="main">

  @yield('content')

<hr>

<div class="well">

  <p>&copy;

    @if (date('Y') > 2015)

      2015 - {{ date('Y') }}
```

Please note: Use the Gist for the code. The Gravatar call is single line:

```
  
        <a href="#"  
            class="dropdown-toggle"  
            data-toggle="dropdown"  
            role="button"  
            aria-haspopup="true"  
            aria-expanded="false">  
  
            {{ Auth::user()->name }}  
  
            <span class="caret"></span>  
  
        </a>  
  
        <ul class="dropdown-menu">  
            <li>  
                <a href="/logout"  
                    onclick="event.preventDefault();  
                    document.getElementById('logout-form').submit();">  
  
                    Logout  
  
                </a>  
  
                <form id="logout-form"  
                    action="/logout"  
                    method="POST"  
                    style="display: none;">  
  
                    {{ csrf_field() }}  
  
                </form>  
  
            </li>  
        </ul>  
    </li>
```

```
<li>
</li>

@else

<li><a href="/login">
    Login
</a>

</li>

<li><a href="/register">
    Register
</a>

</li>

@endif
```

Auth Methods

Laravel ships with some incredibly useful Auth methods, which utilize the Auth facade, right out of the box. Let's look at 3 of them:

```
Auth::check()    //returns true if user logged in, false if not  
  
Auth::id()      // returns the id of the currently logged in user  
  
Auth::user()    // chain the currently logged in user to property &  
                // methods on user model
```

The usefulness of Auth::check() is apparent to us, because we are using it in our new nav. We check to see if the user is logged in:

```
@if (Auth::check())
```

If the user is logged in, we are using the Auth::user() method to get both the name and the email of the user:

```
Auth::user()->name
```

```
Auth::user()->email
```

While Auth::user()->id works, we have a shortcut:

```
Auth::id()
```

If the user is not logged in, we get the else statement:

```
@else

<li><a href="/login">Login</a></li>
<li><a href="/register">Register</a></li>

@endif
```

You can see that this will show the login and registration links.

We also have a bit of javascript and a form post for the logout link:

```
<li>
  <a href="/logout"
    onclick="event.preventDefault();
    document.getElementById('logout-form').submit();">

    Logout

  </a>

  <form id="logout-form"
    action="/logout"
    method="POST"
    style="display: none;">

    {{ csrf_field() }}

  </form>

</li>
```

It's considered best practice to make logout a post request, so we use the javascript snippet to make a form submission. We have to include the csrf_field method as well.

The other big change we did in master.blade.php was to give it a small footer and make the copyright date dynamic:

```
<hr>

<div class="well">

<p>&copy;

@if (date('Y') > 2015)

    2015 - {{ date('Y') }}

@else

    2015

@endif

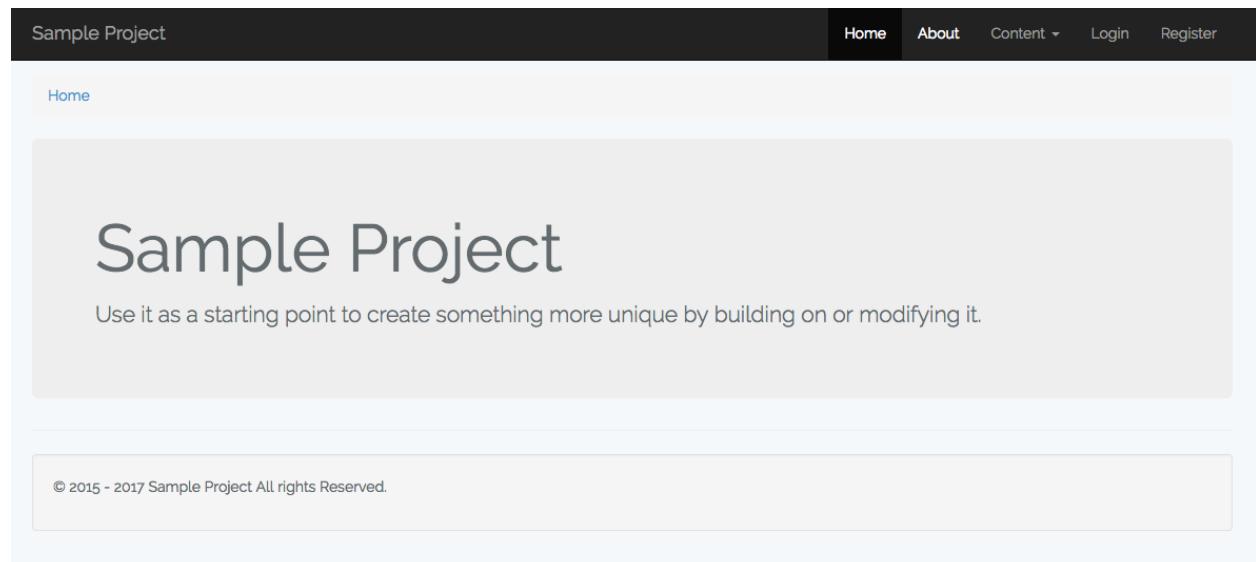
Sample Project All rights Reserved.</p>

</div>
```

I set the first date to 2015, so you can see how we keep the copyright year range up to date.

Keep in mind we are keeping the design as basic as possible to make it easy to decorate later on. We want it professional-looking, but at the same time, bare bones.

Now sample-project.com should look like this:



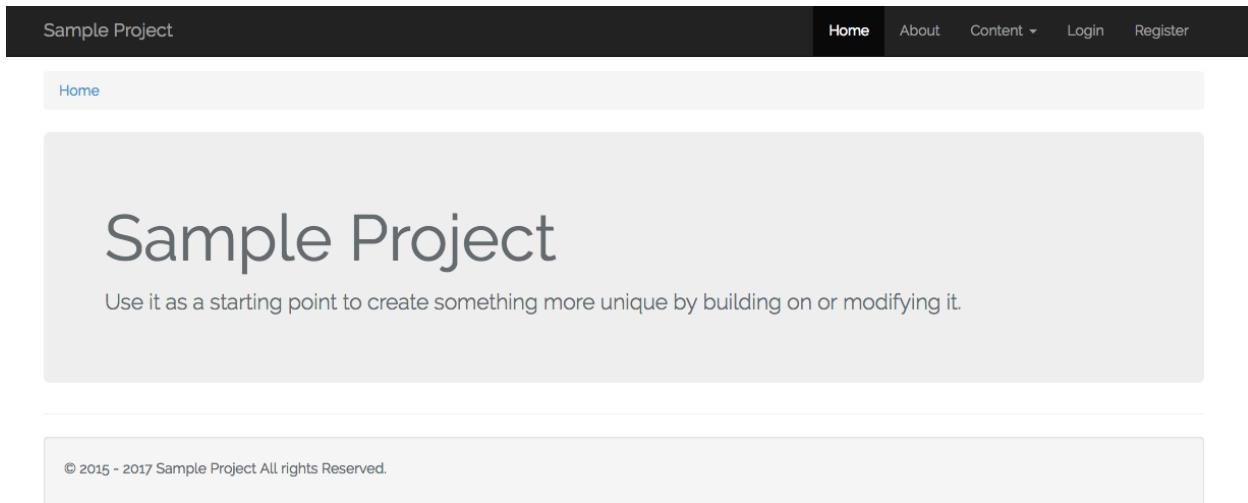
The page is mobile responsive. I'm showing it in the logged out state.

I'm getting a background color to the page that I don't like, so I'm going to drop in a correction, since I already have a spot for it in my css.blade.php file:

```
body{  
    padding-top: 65px;  
    background-color: white;  
}
```

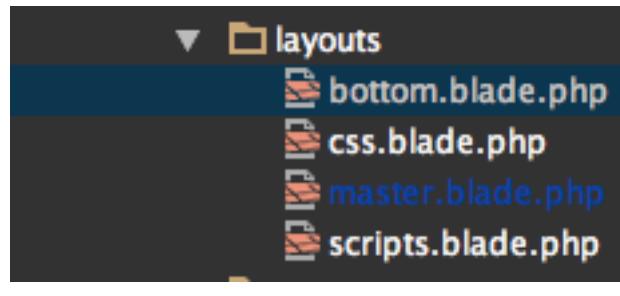
Obviously, we are just adding the one line.

So now our page looks like this:



That makes it clearer and readable. I realize this is all very rudimentary, but this is just meant to be a very basic approach, since this isn't a book about css or bootstrap.

In spite of the improvements to our master.blade.php file, we still have too much for my taste in one file, so let's create another partial to hold the bottom section of the layout. We'll call it bottom.blade.php and place it in the layouts folder. It should look like this:



bottom.blade.php

Cut the following from layouts.master and place it in bottom.blade.php.

Gist:

[bottom.blade.php](#)

From book:

```
<hr>
<div class="well">

    <p>&copy;

    @if (date('Y') > 2015)

        2015 - {{ date('Y') }}

    @else

        2015

    @endif

    Sample Project All rights Reserved.</p>

</div>
```

Then put the following in layouts.master after @yield('content'):

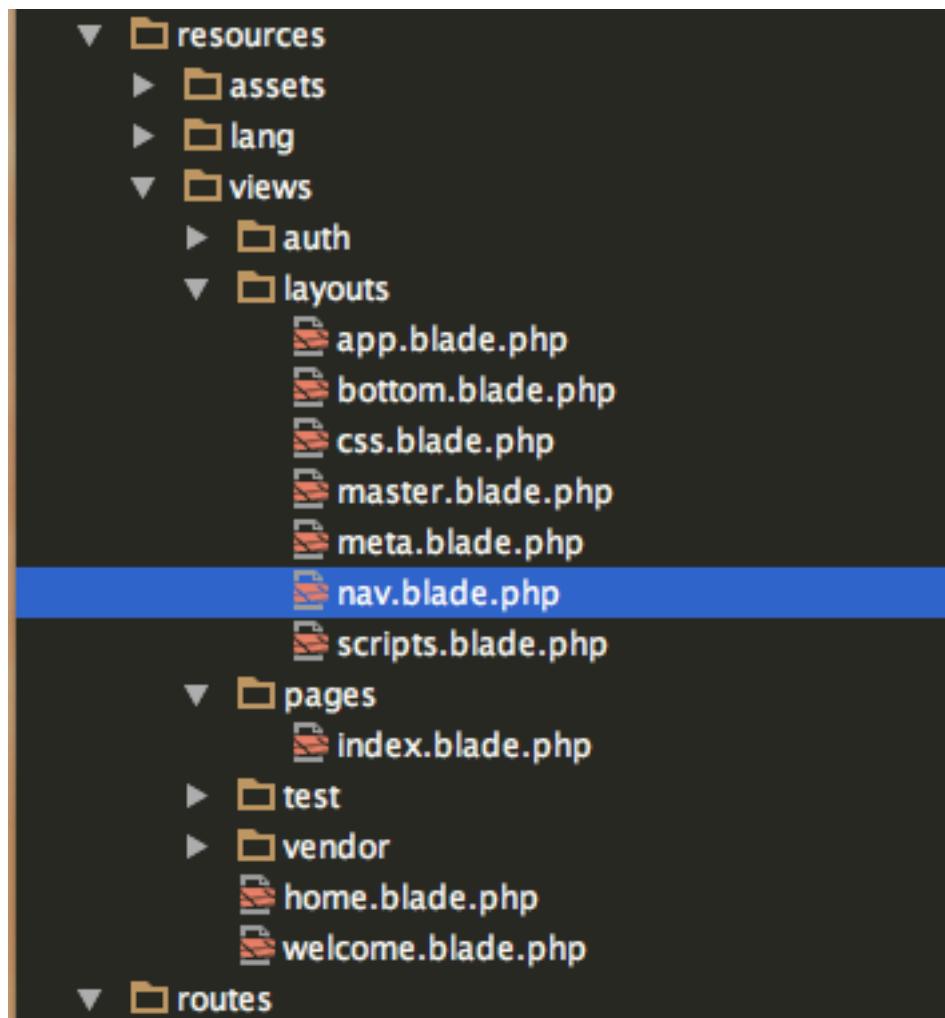
```
@include('layouts.bottom')
```

Now that we're on this method, I'm going to extract out all the remaining parts into partials.

Let's create the following partials in the layouts folder:

- meta.blade.php
- nav.blade.php

You should have the following in your folder:



Let's cut out the meta tags:

Gist:

[meta.blade.php](#)

From book:

```
<meta charset="utf-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1">

<!-- The above 3 meta tags *must* come first in the head; any other
head content must come *after* these tags -->

<meta name="description" content="">
<meta name="author" content="">
```

Make sure to add the include in the master page:

```
@include('layouts.meta')
```

Ok, on to nav.blade.php:

Gist:

[nav.blade.php](#)

From book:

```
<!-- Fixed navbar -->

<nav class="navbar navbar-inverse navbar-fixed-top">

    <div class="container">
        <div class="navbar-header">

            <button type="button"
                class="navbar-toggle collapsed"
                data-toggle="collapse"
                data-target="#navbar"
                aria-expanded="false"
                aria-controls="navbar">
```

```
<span class="sr-only">Toggle navigation</span>
<span class="icon-bar"></span>
<span class="icon-bar"></span>
<span class="icon-bar"></span>

</button>

<a class="navbar-brand"
  href="/">
  Sample Project
</a>

</div>

<div id="navbar" class="navbar-collapse collapse pull-right">

  <ul class="nav navbar-nav">

    <li class="active">
      <a href="/">
        Home
      </a>
    </li>

    <li>
      <a href="#about">
        About
      </a>
    </li>

    <li class="dropdown">
      <a href="#">
```

```
        class="dropdown-toggle"
        data-toggle="dropdown"
        role="button"
        aria-haspopup="true"
        aria-expanded="false">>

    Content

    <span class="caret"></span>

</a>

<ul class="dropdown-menu">

    <li>
        <a href="/widget">
            Widgets
        </a>

        </li>
    </ul>
</li>

@if (Auth::check())

<li class="dropdown">

    <a href="#" 
        class="dropdown-toggle"
        data-toggle="dropdown"
        role="button"
        aria-haspopup="true"
        aria-expanded="false">

        {{ Auth::user()->name }}
    <span class="caret"></span></a>

    <ul class="dropdown-menu">
```

```
<li>

    <a href="/logout"
        onclick="event.preventDefault();
                  document.getElementById('logout-form').submit();">

        Logout

    </a>

    <form id="logout-form"
          action="/logout"
          method="POST"
          style="display: none;">

        {{ csrf_field() }}<br/>

    </form>

</li>

</ul>

</li>

<li>

</li>

@else

<li>

    <a href="/login">

        Login

    </a>

</li>

<li>
```

```
<a href="/register">  
    Register  
</a>  
</li>  
  
@endif  
</ul>  
</div><!--/.nav-collapse -->  
</div>  
</nav>
```

Please reference the Gist for code formatting. Just a heads up, the formatting on views gets especially bad, so it's a good idea to follow along in your IDE or with the Gist.

Let's add the include to our master page:

```
@include('layouts.nav')
```

Finally, our master.blade.php file looks like this.

Gist:

[final master.blade.php](#)

From book:

```
<!DOCTYPE html>

<html lang="en">

<head>

  @include('layouts.meta')

  @yield('title')

  @include('layouts.css')

  @yield('css')

</head>

<body role="document">

  @include('layouts.nav')

<div class="container theme-showcase" role="main">

  @yield('content')

  <hr>

  @include('layouts.bottom')

</div> <!-- /container -->

  @include('layouts.scripts')

  @yield('scripts')

</body>

</html>
```

This makes it really easy for me to digest precedence in the master page. I can clearly see what comes where and in which order. If I want to include custom css that only applies to a specific view, I can inject via the view in the `@yield('css')` section. I can clearly see that as the last css in succession, it will have precedence.

The same is true for scripts that depend on jquery. I have my js call in layouts/scripts, so if I bring in a script that depends on it via the view in the @section('scripts') directive, I can clearly see that it will come after the jquery call.

This order doesn't depend on extracting everything out, but it sure does make it easier to see everything. We also have our code isolated, so if we making a breaking change, for example if we cause an error in the nav, it will be easier to work on and we should know which partial is causing the break.

Some of this is personal preference of course. There's no rule that says you have to extract everything out as I have done. I do it because I have a hard time with large blocks of html code and doing it this way allows me to focus on the separated code.

Ok, so now all we need to do is adjust our views to extend the right master page and then delete the HomeController and the welcome view and app.blade.php. In fact, let's take care of deleting those files while we are thinking of it.

Go ahead and delete:

*HomeController.php *welcome.blade.php. *app.blade.php

We should also remove the /home route from the web.php route file. Remove the following:

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

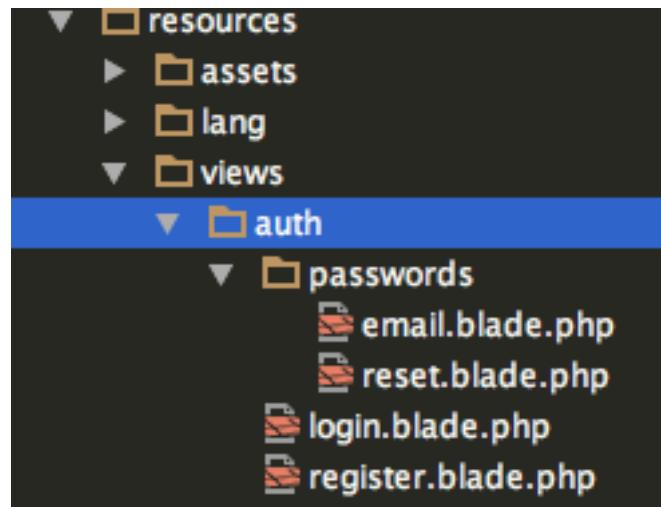
That cleans things up a bit.

Auth Views

Now we can walk through our Auth view files to make sure they extend the correct master page. We will also look at the form elements and how we handle posting the form. We will also be revisiting relevant controller and trait code to become familiar with how the forms are processed.

This will again slow our progress a bit, but don't worry, we won't go too deep into that, just enough to have a basic understanding. Please keep in mind, you don't have to memorize all this.

Here are the files we will be reviewing:



These are the auth views we have already created when we ran `make:auth`. You can see we have an auth folder, and within that a passwords folder. `Login.blade.php` and `register.blade.php` are in the top level auth folder, while `email.blade.php` and `reset.blade.php` are in the passwords folder.

passwords view folder

Let's look inside the passwords view folder. In it, you have `email.blade.php`.

email.blade.php view

This is the form to request the password reset link, which will be sent to the email the user provides.

Since this was an auto-generated file, I won't reproduce it here, I will just give you the replacement:

Gist:

[email.blade.php](#)

From book:

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

<!-- Main Content --&gt;

@section('content')

<div class="container"&gt;

<ol class="breadcrumb"&gt;
<li><a href="/">Home</a></li>
- Send Reset Password</li>
</ol>

<div class="row"&gt;
<div class="col-md-8 col-md-offset-2"&gt;
<div class="panel panel-default"&gt;

<div class="panel-heading"&gt;

Reset Password

&lt;/div&gt;

<div class="panel-body"&gt;

@if (session('status'))

&lt;div class="alert alert-success"&gt;

{{ session('status') }}

&lt;/div&gt;

@endif

&lt;form class="form-horizontal"
role="form"
method="POST"
action="{{ url('/password/email') }}"&gt;

{{ csrf_field() }}

&lt;div class="form-group{{ $errors-&gt;has('email') ? ' has-error' : '' }}&gt;</pre>

```

```
<label for="email"
       class="col-md-4 control-label">

    E-Mail Address

</label>

<div class="col-md-6">

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

    @if ($errors->has('email'))

        <span class="help-block">

            <strong>{{ $errors->first('email') }}</strong>

        </span>

    @endif

</div>
</div>

<div class="form-group">

    <div class="col-md-6 col-md-offset-4">

        <button type="submit"
               class="btn btn-primary">

            <i class="fa fa-btn fa-envelope"></i>

            Send Password Reset Link

        </button>

    </div>
</div>
```

```
</form>

</div>
</div>
</div>
</div>
</div>

@endsection
```

As a reminder, especially for the view files, use the Gists for code formatting. Also note that you might want to wait until after we make all of our changes to our view files in this chapter before testing in the browser, since you will see inconsistent navigation, until we have changed all the @extends directives.

So you can see our first change:

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

We can also add a breadcrumb for navigation, which we will place directly under the container div like so:

```
<div class="container">

<ol class="breadcrumb">
    <li><a href="/">Home</a></li>
    <li>Send Reset Password</li>
</ol>
```

Those are the only changes we need to make to the file. But let's look at some of the more interesting parts in the file.

We can see there is an if statement, using blade's @if syntax. If the session has a 'status' then we utilize the following:

```
@if (session('status'))  
  
    <div class="alert alert-success">  
  
        {{ session('status') }}  
  
    </div>  
  
@endif
```

If the reset form is successfully posted, then it flashes a success message back to the page via the session. If you are interested in how that happens, go back and look at the sendResetResponse method in the ResetsPasswords trait at:

`vendor/laravel/framework/src/Illuminate/Foundation/Auth/ResetsPasswords.php`

Next we will look at the first part of the form:

```
<form class="form-horizontal"  
      role="form"  
      method="POST"  
      action="{{ url('/password/email') }}>  
  
    {{ csrf_field() }}
```

Here you can see we are setting the action using the `url` helper. Putting the double braces around it will make blade echo out the value, in our case:

`http://sample-project.com/password/email`

In the case of the `csrf_field()` method, this will echo out a hidden form field with csrf token:

```
<input type="hidden"  
      name="_token"  
      value="JJ8ElyCN0UI0m3FyHUJqVXGBiMVz1gLGFvR6Ve7">
```

We use this to prevent forged requests to our application.

Moving on to the form elements:

```
<div class="form-group{{ $errors->has('email') ? ' has-error' : '' }}>  
  
<label class="col-md-4 control-label">E-Mail Address</label>  
  
<div class="col-md-6">  
  
  <input type="email"  
        class="form-control"  
        name="email"  
        value="{{ old('email') }}>  
  
  @if ($errors->has('email'))  
  
    <span class="help-block">  
      <strong>{{ $errors->first('email') }}</strong>  
    </span>  
  
  @endif  
  
</div>  
</div>
```

The \$errors variable, which holds validation errors from the request, is always available to the view, so we just have to access the key to get the value of the message, in this case, ‘email’. In the following div, we check for an error to determine which class name we should use on the div:

```
<div class="form-group{{ $errors->has('email') ? ' has-error' : '' }}>
```

This uses a simple ternary to add has-error to the div class, so if you are displaying an error message, you get the right style. It's a nice way of doing it because you get the error printed line by line.

Then we are using the @if statement to check to see if there are errors in the request that have been fed back to the user and display them:

```
@if ($errors->has('email'))  
  
    <span class="help-block">  
  
        <strong>{{ $errors->first('email') }}</strong>  
  
    </span>  
  
@endif  
  
</div>  
</div>
```

Finally, we get the submit button:

```
<div class="form-group">  
  
    <div class="col-md-6 col-md-offset-4">  
  
        <button type="submit"  
               class="btn btn-primary">  
            <i class="fa fa-btn fa-envelope"></i>  
  
            Send Password Reset Link  
  
        </button>  
  
    </div>  
  
</div>
```

There's nothing special about the submit button, except that it's using an icon from Font-Awesome. We haven't pulled that in yet, so let's add that to our css.blade.php file.

We are going to use a CDN, so let's grab the latest:

[MaxCDN](#)

Even though the call is displayed here as multiple lines, add it as a single line in our css.blade.php file:

```
<!-- Styles -->

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

<link rel="stylesheet"
      href=
      "https://
      maxcdn.bootstrapcdn.com/font-awesome/4.5.0/css/font-awesome.min.css">

<style>

body{
    padding-top: 65px;
    background-color: white;
}

.circ{

    width : 40px;
    vertical-align: middle;
    border-radius: 50%;
    box-shadow: 0 0 0 2px #fff;
    margin-top: 5px;
}

.circ:hover{

    box-shadow: 0 0 0 3px #f00;
}
```

```
}
```

```
</style>
```

We're only adding the one line, so no gist.

Obviously I had to break up the lines to avoid word-wrap, so make sure it's all one line where appropriate.

We're going to be reorganizing our css assets later in the book, but for now we're keeping it super simple.

reset.blade.php

Moving on in the passwords view folder, we have the reset.blade.php view. Here is the new version:

Gist:

[reset.blade.php](#)

From book:

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

@section('content')



- Home
- Submit Password Reset



Reset Password


```

```
<div class="panel-body">

<form class="form-horizontal"
      role="form"
      method="POST"
      action="{{ url('/password/reset') }}>

    {{ csrf_field() }}

    <input type="hidden"
           name="token"
           value="{{ $token }}>

    <div class="form-group{{ $errors->has('email') ? ' has-error' : '' }}>

        <label for="email"
              class="col-md-4 control-label">
            E-Mail Address
        </label>

        <div class="col-md-6">

            <input id="email"
                  type="email"
                  class="form-control"
                  name="email"
                  value="{{ $email or old('email') }}" autofocus>

            @if ($errors->has('email'))

                <span class="help-block">
                    <strong>{{ $errors->first('email') }}</strong>
                </span>

            @endif

        </div>
    </div>
```

```
<div class="form-group{{ $errors->has('password') ? ' has-error' : '' }}>

<label for="password"
      class="col-md-4 control-label">

    Password

</label>

<div class="col-md-6">

<input id="password"
       type="password"
       class="form-control"
       name="password">

@if ($errors->has('password'))

  <span class="help-block">

    <strong>{{ $errors->first('password') }}</strong>

  </span>

@endif

</div>
</div>

<div class="

  form-group {{ $errors->has('password_confirmation') ? ' has-error' : '' }}"

>

<label for="password-confirm"
      class="col-md-4 control-label">

    Confirm Password

</label>

<div class="col-md-6">
```

```
<input id="password-confirm"
       type="password"
       class="form-control"
       name="password_confirmation">

@if ($errors->has('password_confirmation'))

<span class="help-block">

<strong>{{ $errors->first('password_confirmation') }}</strong>

</span>

@endif

</div>
</div>

<div class="form-group">

<div class="col-md-6 col-md-offset-4">
    <button type="submit"
            class="btn btn-primary">

        Reset Password

    </button>
</div>
</div>

</form>

</div>
</div>
</div>
</div>
</div>

@endsection
```

We again changed the @extends line to:

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

And also add the breadcrumb:

```
<div class="container">

<ol class="breadcrumb">
    <li><a href="/">Home</a></li>
    <li>Submit Password Reset</li>
</ol>
```

So in looking at the whole file, you can see that this is the form that actually submits the new password. We can see that it utilizes a hidden field to hold the token that it needs to allow the request to be processed:

```
<input type="hidden" name="token" value="{{ $token }}">
```

The \$token was passed from the email link to the PasswordController, where it uses the ResetsPasswords trait located at:

vendor/laravel/framework/src/Illuminate/Foundation/Auth/ResetsPasswords.php

It uses the showResetForm method in the trait to pass the token to this view.

```
public function showResetForm(Request $request, $token = null)
{
    return view('auth.passwords.reset')->with([
        'token' => $token,
        'email' => $request->email
    ]);
}
```

A good IDE will allow you to click into the trait to display the file, but I keep listing the path in case you need it for reference.

Ok, back to the view file. Now when you look over this form, you may wonder how we handle validation on password confirmation. You can see that we have a password confirmation field on the form, with a name of password_confirmation, but there is no corresponding field for validation in the trait. Laravel is so smart that it knows that if one of your fields is something_confirmation, it handles it a certain way, as long as you are using the validation rule properly. Let's take a look:

Inside the ResetsPasswords trait, you can see this in action in the reset method:

```
$this->validate($request, $this->rules(), $this->validationErrorMessages());
```

The rules method is being called and that has our validation rules:

```
protected function rules()
{
    return [
        'token' => 'required',
        'email' => 'required|email',
        'password' => 'required|confirmed|min:6',
    ];
}
```

In between the pipe characters on the 'password' line is the word 'confirmed'.

You can see this also in the RegisterController in the validator method, which we use when we create a user::

```
'password' => 'required|min:6|confirmed',
```

We use the pipe symbol to separate the validation rules, and if it has confirmed like in the snippet above, it will look for a field named password_confirmation as a concatenation of the key, in this case, ‘password’ and the string ‘_confirmation’. It will return an error if it does not find it or they do not match. Now that’s a lot of functionality rolled up into almost no code. It’s just awesome. It’s a minor thing but it shows just how much attention to detail has gone into Laravel.

We are going to cover validation in more detail when we work on how we handle requests.

Auth View Folder

register.blade.php

We’re going to replace what is there with the following:

Gist:

[register.blade.php](#)

From book:

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

@section('content')

<div class="container">

    <ol class="breadcrumb">
        <li><a href="/">Home</a></li>
        <li>Register</li>
    </ol>

    <div class="row">

        <div class="col-md-8 col-md-offset-2">

            <div class="panel panel-default">
                <div class="panel-heading">Register</div>
                <div class="panel-body">
```

```
<form class="form-horizontal"
      role="form"
      method="POST"
      action="{{ url('/register') }}>

{{ csrf_field() }}

<div class="form-group{{ $errors->has('name') ? ' has-error' : '' }}>

<label for="name"
       class="col-md-4 control-label">

    Name

</label>

<div class="col-md-6">

<input id="name"
       type="text"
       class="form-control"
       name="name"
       value="{{ old('name') }}" autofocus>

@if ($errors->has('name'))

<span class="help-block">

<strong>{{ $errors->first('name') }}</strong>

</span>

@endif

</div>
</div>

<div class="form-group{{ $errors->has('email') ? ' has-error' : '' }}>

<label for="email"
       class="col-md-4 control-label">

    E-Mail Address

</label>
```

```
</label>

<div class="col-md-6">

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

@if ($errors->has('email'))

<span class="help-block">

<strong>{{ $errors->first('email') }}</strong>

</span>

@endif

</div>
</div>

<div class="form-group{{ $errors->has('password') ? ' has-error' : '' }}>

<label for="password"
      class="col-md-4 control-label">

    Password

</label>

<div class="col-md-6">

<input id="password"
       type="password"
       class="form-control"
       name="password">

@if ($errors->has('password'))

<span class="help-block">
```

```
<strong>{{ $errors->first('password') }}</strong>

</span>

@endif

</div>
</div>

<div class="

    form-group{{ $errors->has('password_confirmation') ? ' has-error' : '' }}

">

<label for="password-confirm"
      class="col-md-4 control-label">

    Confirm Password

</label>

<div class="col-md-6">

<input id="password-confirm"
      type="password"
      class="form-control"
      name="password_confirmation">

@if ($errors->has('password_confirmation'))

<span class="help-block">

<strong>{{ $errors->first('password_confirmation') }}</strong>

</span>

@endif

</div>
<div>

<div class="form-group">
```

```
<div class="col-md-6 col-md-offset-4">

<button type="submit"
        class="btn btn-primary">

    Register

</button>

</div>
</div>
</form>

</div>
</div>
</div>
</div>
</div>

@endsection
```

Again, I highly recommend using the gist.

Note the change @extends to:

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

Next we have the breadcrumb under the container div(shown for reference):

```
<div class="container">

<ol class="breadcrumb">
    <li><a href="/">Home</a></li>
    <li>Register</li>
</ol>
```

So most of the registration form we have covered before. You can see in the form action it's a post request and we are using the url helper again:

```
<form class="form-horizontal"
      role="form"
      method="POST"
      action="{{ url('/register') }}>
```

You can see we are posting to '/register,' which is a URI that the Route::auth() method in the web.php file will recognize and route to the RegisterController, which will use the RegistersUsers trait to process it.

We again have the csrf method:

```
{{ csrf_field() }}
```

So one thing we haven't had before is the use of the global old function, which allows us to retrieve input from the session:

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

So if the form fails because of password or email input, they will not have to enter their name again. This makes for friendly UI that users will appreciate. Laravel makes doing this incredibly easy.

login.blade.php

Gist:

[login.blade.php](#)

From book:

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

@section('content')

<div class="container">
    <ol class="breadcrumb">
        <li><a href="/">Home</a></li>
        <li>Login</li>
    </ol>

    <div class="row">
        <div class="col-md-8 col-md-offset-2">

            <div class="panel panel-default">
                <div class="panel-heading">Login</div>
                <div class="panel-body">

                    <form class="form-horizontal"
                          role="form"
                          method="POST"
                          action="{{ url('/login') }}>

                        {{ csrf_field() }}
```

```
@if ($errors->has('email'))  
  
    span class="help-block">  
  
        strong{{ $errors->first('email') }}  
  
      
  
@endif  
  
</div>  
</div>  
  
<div class="  
  
    form-group{{ $errors->has('password') ? ' has-error' : '' }}  
  
">  
  
<label for="password"  
      class="col-md-4 control-label">  
  
    Password  
  
</label>  
  
<div class="col-md-6">  
  
<input id="password"  
      type="password"  
      class="form-control"  
      name="password">  
  
@if ($errors->has('password'))  
  
    span class="help-block">  
  
        strong{{ $errors->first('password') }}  
  
      
  
@endif
```

```
</div>
</div>

<div class="form-group">
<div class="col-md-6 col-md-offset-4">
<div class="checkbox">

<label>

<input type="checkbox"
       name="remember">

    Remember Me

</label>

</div>
</div>
</div>

<div class="form-group">
<div class="col-md-8 col-md-offset-4">

<button type="submit"
        class="btn btn-primary">

    Login

</button>

<a class="btn btn-link"
   href="{{ url('/password/reset') }}">

    Forgot Your Password?

</a>

</div>
</div>
</form>

</div>
</div>
</div>
```

```
</div>
</div>

@endsection
```

So we changed the @extends and added the breadcrumb.

Looking over the entire file, most of the login view we have seen before. We do have a checkbox for remember, which the user will see as remember me.

Inside of the AuthenticatesUsers trait:

```
vendor/laravel/framework/src/Illuminate/Foundation/Auth/AuthenticatesUsers.php
```

We look for the remember field inside of the attemptLogin method in the following line:

```
return $this->guard()->attempt(
    $this->credentials($request), $request->has('remember')
);
```

The \$request->has method is returning true or false, so that's how we know if we need to set the cookie that remembers a user.

Change redirectTo Properties

If you have not already done so, you need to change the redirectTo property on both LoginController and RegisterController to the following:

```
protected $redirectTo = '/';
```

That will return the user to the '/' uri when logging in or registering, which is what we want. You'll know the user is logged in because the user name will appear in the top right nav, along with the gravatar.

And with that, our review of our auth views is complete. You can play around with the login and registration and all the navigation should be consistent.

Summary

By using the artisan make:auth command, we quickly stood up our auth views, which allows us to utilize the Auth and Password Controllers and their traits. Along the way, we learned a little about the use of traits, a subject we will be returning to later in the book.

We also now have a working user model, allowing us to register and login to the application. It took us a while to get here, but we have been learning a lot of interesting bits and pieces of the framework along the way, including some of the password recovery methods.

Don't worry if you don't have everything committed to memory, it takes a long time when you are just learning to digest all this information. You are slowly building your skills and your knowledge in Laravel. We will continue strengthen them in the next chapter.

Chapter 6: Working with the RESTful Pattern

We're going to be working with the RESTful pattern in this chapter. We will set up a model, migration, route resource, RESTful controller, and matching views. We're going to do it in a way so that it acts as a prototype for future models, meaning we can refer to this code for future implementations.

But before we move into that, let's do a quick implementation of a Sweet Alert package for flash messaging. Jeffery Way of [Laracasts.com](#) has a great video on how to build a global helper for Sweet Alert and that's where I first saw it being used. But rather than build the helper, we will simply use the package available, which is easier to set up.

As I mentioned before, we are going along mimicking workflow, so this is where I would be adding Sweet Alert as part of a general setup of the application, before digging into the first new model.

Sweet Alert

This is a package written by Socieboy, who you can find on Github. Here is the page for his package:

[SocieBoy Sweet Alert](#)

What this does is create a nicely formatted alert for your application, so you can give the user a flash message when things go right or wrong. I will demonstrate this after we have the package installed.

Let's pull this in by doing the following steps:

First, let's pull in the package from Socieboy. Normally I would run composer require, but that didn't work with this project, it hasn't been updated in a while. So we need to add it to composer.json in the require section:

```
"socieboy/alerts" : "dev-master"
```

Then we will run composer update from the command line:

```
composer update
```

The difference between composer install and composer update is that composer install will not update versions in your require section of your composer.json file, it will just make sure you have the required versions. This is a safer way to add a single package.

The safest and easiest way to add a package is with the composer require command, but like I said that was returning an error, so we couldn't use it.

Okay, once that is installed, go to config/app.php and add the following to the providers array in the Package Service Providers section:

```
Socieboy\Alerts\AlertServiceProvider::class,
```

The next step is to publish the vendor assets. This will create the view partials needed to run the alerts. We do this with an Artisan command:

```
php artisan vendor:publish --tag=alerts
```

This will create the css file and the js file that you need in their respective folders.

In the layouts folder, in your css.blade.php file, place the following after the call to app.css:

```
<!--sweet alert-->  
<link rel="stylesheet" type="text/css" href="/css/sweetalert.css">
```

Next, at the bottom of your scripts.blade.php file, place the following:

```
<!--sweet alert-->  
<script src="/js/sweetalert.js"></script>
```

Make sure that goes after your app.js file.

For production, you can download the minified assets instead from:

<http://t4t5.github.io/sweetalert/>

Just remember to adjust your calls to the files to reflect that if you use the .min files.

Finally, on master.blade.php, let's put @include('Alerts::show') in between @include('layouts.scripts') and @yield('scripts'). I'm showing all 3 lines for reference, but you only need to add the one line:

```
@include('layouts.scripts')

@include('Alerts::show')

@yield('scripts')
```

And that's all there is to it. Hopefully you are seeing the benefits of how the partials keep everything separated, more focused, and easier to work with.

Note the Alerts::show call in the include is to the namespace Alerts and the show view within it, all of which is in the vendor folder socieboy.

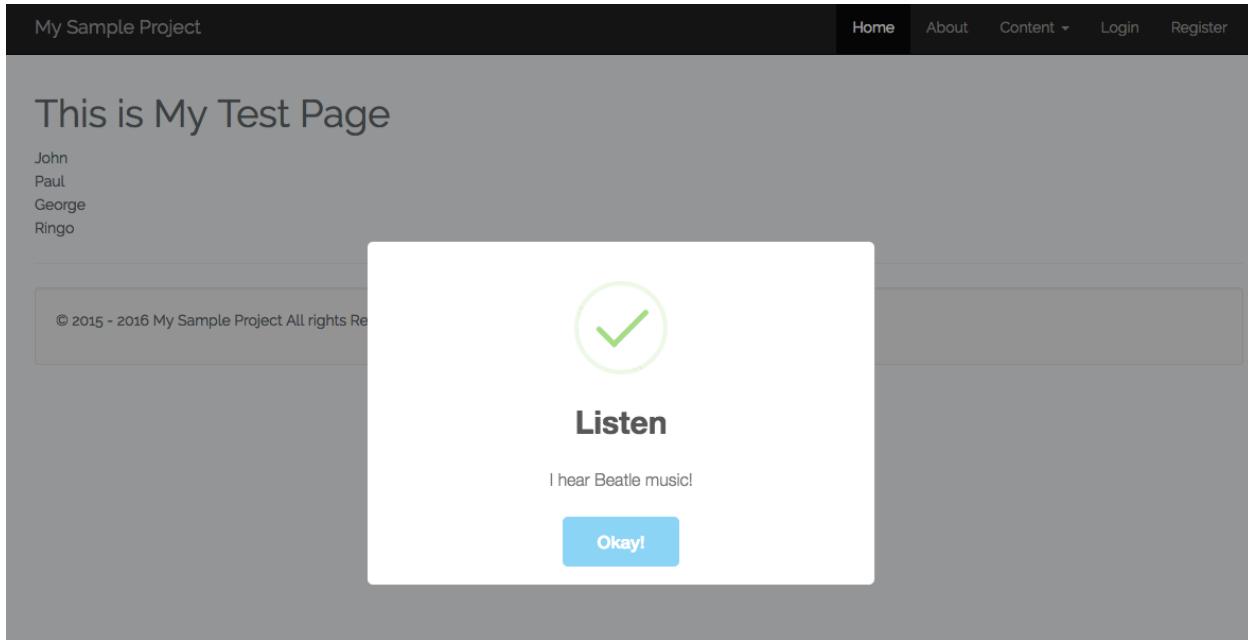
Now to test this, it's pretty easy. We already have a test controller setup. So all we have to do is change the index method to the following:

```
public function index()
{
    $beatles = ['John', 'Paul', 'George', 'Ringo'];

    alert()->overlay('Listen', 'I hear beatle music!', 'success');

    return view('test.index', compact('beatles'));
}
```

That doesn't really mean anything, but it will cause an overlay with a success message that looks like this:



Let's do one more example. Replace the alert overlay in the TestController with the following:

```
alert()->error('Problem', 'Cannot hear');
```

And you will see that returns an error alert.

We are following the format directly from the [github instructions](#) Socieboy provides there:

Usage

On your controllers is a perfect place to use it, any way you can fire the alerts from jobs or events.

```
alert('Title', 'Message')

alert()->error('Title', 'Message')

alert()->success('Title', 'Message')

alert()->overlay('Title', 'Message')
```

Override the type of overlay

```
alert()->overlay('Title', 'Message', 'error')

// success (default)

// error

// info
```

If you want the overlay to stay visible until the user clicks ok, you can do something like:

```
alert()->overlay('Problem', 'Cannot hear', 'error');
```

So you can see the instructions are clear. I only showed these to you so you could see how simple this is. You should refer to the above Github page for the actual instructions, since they may get updated before I can update the book.

I'm not endorsing Sweet Alert, but it is pretty easy to use and spices up the presentation a bit. Now you know how you can provide the user with feedback on their actions, which makes for more engaging UI.

Flash Messages

If you want to send flash messages without a package like Sweet Alert, you can reference the laravel docs:

[flash messages](#)

In your controller, you would do something like:

```
session()->flash('status', 'Task was successful');
```

The session() method is available globally to the controller and the views. In this example, 'status' is the key and 'Task was successful' is the value.'

Then in your view, you would access it like so:

```
@if(session()->has('status'))  
    {{ session('status') }}  
  
@endif
```

Play around with that in your test controller and test index view, you will see how it works. You can then style the output in a bootstrap alert, if you wanted to go in that direction.

Ok, so now we're ready to move on to making a model.

Model

We going to start by creating a model that won't actually get used on your eventual site. We're building it as a prototype that we can refer to.

Without getting into a long technical explanation of what the Model is in terms of the MVC pattern, which would have no useful purpose here, let's talk about it in practical terms of Laravel. The short version is that the model is responsible for interacting with the database.

We already have the User model that Laravel provided us with, so we can store users in the database, among other things. Our other models will tend to be less complicated because we don't typically have to encrypt the data, like we do when we're dealing with passwords.

We're not so concerned right now about how useful our model is, we're more concerned with how to create one and how it works. So let's just jump in and build one, and most of it will become abundantly clear through workflow.

Let's start by going to the command line and running:

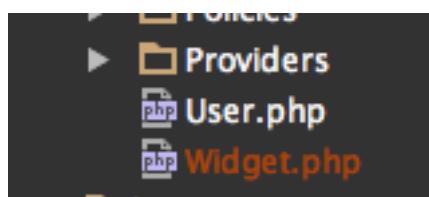
```
php artisan make:model Widget -m
```

We're naming our model Widget, which is just a generic name that can represent anything. You can see that we added the -m flag, and that will simultaneously create a migration along with the model. This is excellent workflow efficiency.

After running the command, you should see confirmation of both files in the command line like so:

```
Bills-MacBook-Pro:my-sample-project billk$ php artisan make:model Widget -m  
Model created successfully.  
Created Migration: 2016_08_20_171341_create_widgets_table  
Bills-MacBook-Pro:my-sample-project billk$
```

Since there is no models folder in Laravel, Widget.php will go directly into the app folder. You can find it next to your User model here:



Note: Let me reiterate, it's not in the Providers folder, it's in the app folder. Also, when you create a file by using artisan, git does not know about it and you have to tell git about it. That's why the file is highlighted in red above. Use whatever method your IDE provides for adding it to your version control, if you are using version control, which is highly recommended.

Now let's look at the new model file:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Widget extends Model
{
    //

}
```

When I first saw this myself, I thought it was an empty stub, but it's not. Much of the model logic is extracted out to the framework, so it makes working with it clean and easy. We will be adding to what you see here, but most of the heavy lifting is already done for us in the Active Record implementation.

Laravel has a name for its Active Record implementation. It's called Eloquent and it gives us a highly intuitive syntax for doing common tasks with the DB. We will see that in action later.

For now let's go to the other file we created, the migration. You can find that in database/migrations and will look like this:

```
<?php

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

class CreateWidgetsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */

    public function up()
    {
        Schema::create('widgets', function (Blueprint $table) {

            $table->increments('id');
            $table->timestamps();

        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */

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

You can see that artisan has done quite a lot of work for us. It named the table for us, widgets, which is plural of course. The convention is singular for model, plural for table name. It's also giving us an auto-increment id column and a method that will create both created_at and updated_at datetime columns.

You can see it also gives us the down method to drop the table:

```
Schema::dropIfExists('widgets');
```

Now since the Widget model doesn't serve any working purpose in the application, we are going to keep it extremely simple. All we are going to do is add a name column to the up method:

```
$table->string('name')->unique();
```

When you use the string method, the default is a varchar column with 255 characters, so once migrated the type on the column will look like:

```
varchar(255)
```

You can of course specify how many characters you want to allow as the second argument. That would look like this:

```
$table->string('name', 60)->unique();
```

So in that case it would limit it to 60 characters. But let's stick with the default for now.

Also note, that we put a unique constraint on it. So now your up method should look like this:

```

public function up()
{
    Schema::create('widgets', function (Blueprint $table) {

        $table->increments('id');
        $table->string('name')->unique();
        $table->timestamps();

    });
}


```

I'm not going to provide a gist, since you only have to add the one line.

Now let's run the migration from the command line:

```
php artisan migrate
```

And with that we have our widgets table. You can check it out in PhpMyAdmin:

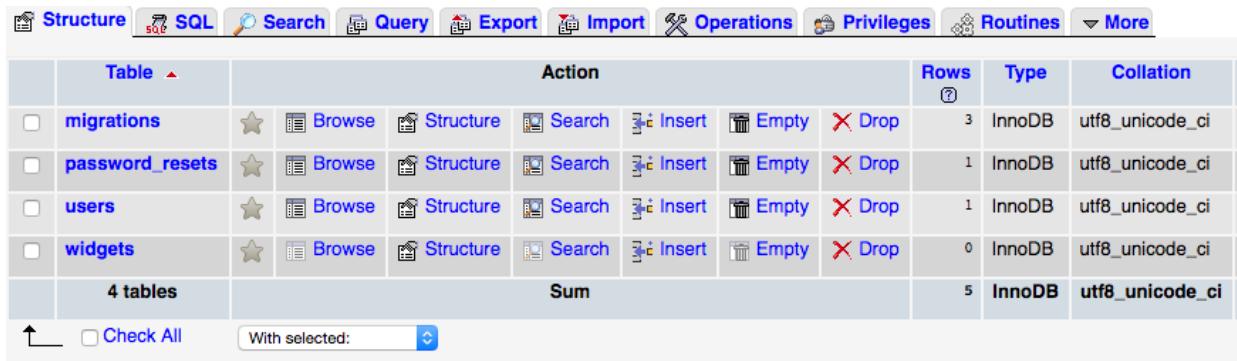


	Table	Action							Rows	Type	Collation
<input type="checkbox"/>	migrations							3	InnoDB	utf8_unicode_ci	
<input type="checkbox"/>	password_resets							1	InnoDB	utf8_unicode_ci	
<input type="checkbox"/>	users							1	InnoDB	utf8_unicode_ci	
<input type="checkbox"/>	widgets							0	InnoDB	utf8_unicode_ci	
4 tables		Sum							5	InnoDB	utf8_unicode_ci

Up Check All With selected:

Now let's return to the Widget model.

Since we're planning to add records to the widgets table, we need to add the following property to our Widget model:

```
/*
 * The attributes that are mass assignable.
 *
 * @var array
 */

protected $fillable = ['name'];
```

This tells Eloquent that this is a mass assignable property, and if we didn't have this, Laravel's Eloquent wouldn't allow us to save any records. Eloquent is the perfect name and you will understand why when we see it in action later.

Back to our fillable property. You'll have to add to the fillable array every time you create a new column on a table that you want to save data to. Note that we don't need to specify id, since it's auto-increment, and we also don't need to include the timestamps, they are also already included in the model.

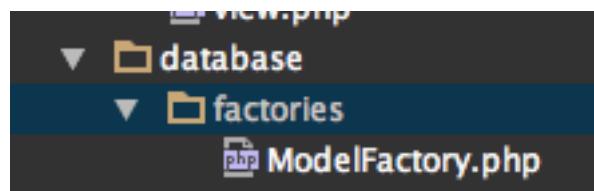
So from a typical workflow standpoint, you could move onto to creating the routes, controller and views for our model. If you follow that pattern, you will make a model with migration, and then create the routes using a route resource. You could follow that by using artisan to make the controller, then work on the matching views and controller actions individually. I follow that workflow a lot.

Before we move on to demonstrating that, let's look at a really cool feature that Laravel has for making test data.

Model Factory

We're going to look at 2 ways to use the Model factory that ships with Laravel to create test records that we can play with for our Widget model.

Let's open database/factories/ModelFactory.php. You can find it here:



You can see that it comes with a factory method to seed the users table:

```
$factory->define(App\User::class, function (Faker\Generator $faker) {  
  
    return [  
  
        'name' => $faker->name,  
        'email' => $faker->safeEmail,  
        'password' => bcrypt(str_random(10)),  
        'remember_token' => str_random(10),  
  
    ];  
  
});
```

Let's add a new factory method to the file below the one for user:

```
$factory->define(App\Widget::class, function ($faker) {  
  
    return [  
  
        'name' => $faker->unique()->word,  
  
    ];  
  
});
```

To populate our name column in our widgets table, we are going to use a faker word, making sure that it's unique, since we have that constraint on our db column.

You can check with the [fzaninotto/Faker Git page](#) for a full set of formatters you can play with, it's pretty awesome.

Inside the factory method, we hand in the appropriate model.

Then inside of database/seeds, you will find DatabaseSeeder.php, in which we will add the following use statement:

```
use App\Widget;
```

And also change the run method to the following:

```
public function run()
{
    Widget::unguard();

    Widget::truncate();

    factory(Widget::class)->create();

    Widget::reguard();
}
```

You can see we truncate the existing records, then call the factory method, which will create a single record. If we want more records, we can specify that as the second parameter:

```
factory(Widget::class, 30)->create();
```

In this case we would get 30 records. But let's step back and just create a single record by running the following from the command line:

```
php artisan db:seed
```

Now if you want to get rid of the records easily, you can:

```
php artisan migrate:rollback
```

But that gets messy if you have already done other migrations, so not recommended for most cases.

In fact, I don't like bothering with db:seed at all. I find it's much easier to run the factory method straight from an artisan tinker command.

Let's check out how it works for this by running the following from the command line:

```
php artisan tinker
```

That will bring up the tinker prompt, which looks like this:

```
Bills-MacBook-Pro:sample-project billk$ php artisan tinker
Psy Shell v0.8.1 (PHP 5.6.7 - cli) by Justin Hileman
>>> |
```

Ok, so first we want to truncate the widgets table, and we can do that with the following command:

```
\App\Widget::truncate();
```

The response we are expecting looks like this:

```
>>> \App\Widget::truncate();
=> Illuminate\Database\Eloquent\Builder {#680}
>>> |
```

So that gets rid of the previous records. If you check your widgets table in your db, it should be empty now.

So now what we're going to do is see the table directly from tinker with the following command:

```
factory('App\Widget', 30)->create();
```

This will return a list of models it created:

```
        id: 24,
    },
    App\Widget {#737
        name: "aspernatur",
        updated_at: "2016-08-20 21:38:53",
        created_at: "2016-08-20 21:38:53",
        id: 25,
    },
    App\Widget {#738
        name: "incidunt",
        updated_at: "2016-08-20 21:38:53",
        created_at: "2016-08-20 21:38:53",
        id: 26,
    },
    App\Widget {#739
        name: "fugit",
        updated_at: "2016-08-20 21:38:53",
        created_at: "2016-08-20 21:38:53",
        id: 27,
    },
    App\Widget {#740
        name: "at",
        updated_at: "2016-08-20 21:38:53",
        created_at: "2016-08-20 21:38:53",
        id: 28,
    },
    App\Widget {#741
        name: "non",
        updated_at: "2016-08-20 21:38:53",
        created_at: "2016-08-20 21:38:53",
        id: 29,
    },
    App\Widget {#742
        name: "modi",
        updated_at: "2016-08-20 21:38:53",
        created_at: "2016-08-20 21:38:53",
        id: 30,
    },
],
}
>>> 
```

And if you check in your DB, you will see they have been persisted.

I find this is a much more convenient way to seed the db, no class to change, etc.

To terminate your tinker session, type Control D (Control C for Windows) and press enter from the command line:



```
>>>
Exit: Ctrl+D
Bills-MacBook-Pro:my-sample-project billk$
```

Now when you have the faker records in already in your table, and you try to run the factory method to seed them a second time, you may get an error due to the unique constraint. That's because faker doesn't know what is already in your db. So if you need to run the test again, truncate the table first, then you can run it again.

Anyway, now that this is done, we have 30 widget records to work with.

Route Resource

Ok, we're ready to move on and create a route resource for our widget model. So let's start by adding the following line to your routes/web.php file:

```
Route::resource('widget', 'WidgetController');
```

This will create all the restful routes that we need for our widget model.

So at this point, your routes file should look like this:

```
<?php

/*
-----
/ Web Routes
-----
/
| Here is where you can register web routes for your application. These
| routes are loaded by the RouteServiceProvider within a group which
| contains the "web" middleware group. Now create something great!
|
*/

// Auth routes

Auth::routes();

// home page route

Route::get('/', 'PagesController@index');

// test route

Route::get('test', 'TestController@index');

// Widget routes

Route::resource('widget', 'WidgetController');
```

You can see I placed a comment above each route and that the routes are ordered alphabetically. As your web.php file grows, this makes it easier to work with your routes.

RESTful Controller

Ok, on to creating the controller. We're going to use the artisan command to make our controller, and because we are adding the --resource flag, it will give us the 7 RESTful methods by default:

```
php artisan make:controller WidgetController --resource
```

Now if you look in app/Http/Controllers, you should see your WidgetController.php file. If you open that, you can see your method stubs. We already covered this ground when we built our Test and Pages controllers.

The workflow question at this point is what comes next. Since we have test records in the DB thanks to Faker and our model factory test, we could work on the index method first.

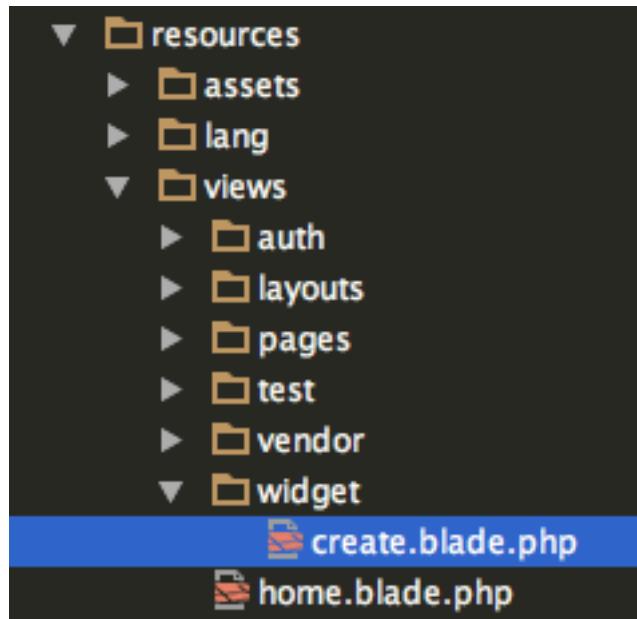
But that's not typically how I do it and here's why. I start with the create form, then I make the create method on the controller, which is typically one line just to return the form view, then I work on the store method, the show method, and finally the index view.

The reason I do it this way is that it gives me a chance to make adjustments to the data structure as I go along. In practice, I find myself making little tweaks as I go through those methods, so this makes a lot of sense as a workflow pattern. Then after I feel I've gotten it right, I create the index method and view, which will typically show all records.

With our Widget model, the data structure is so simple, we actually could start with the index method first, but we will begin by modifying the create method on the controller. Let's just add the one line, so the whole thing looks like this:

```
public function create()
{
    return view('widget.create');
}
```

You know by now that it will look for a widget folder within the views folder and a create view within that. We don't have either, so let's make our widget folder in views now. It should look like this:



Within the widget folder, make a file named create.blade.php and leave it blank for now.

Errors List

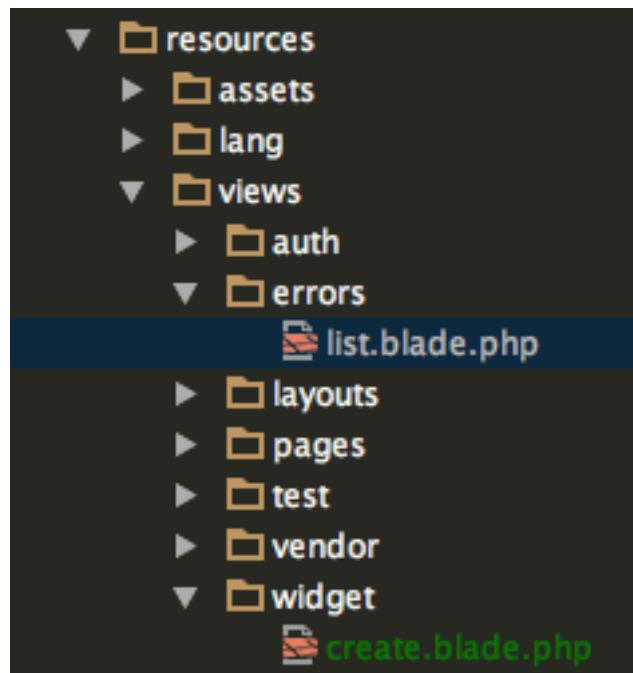
I mentioned in an earlier chapter that from the controller, we always have access to an \$errors variable that we can use in our views. This allows us to give error feedback to the users on the forms we are creating.

Rather than account for this every time we do a form, we could create a partial that we can include, so we only have to write the code once. I'm going to show you how to do this, but we won't be using it until much later in the book.

Errors Folder

We are going to need an errors folder anyway because we will want to provide a pretty response on many different types of exceptions. So let's go ahead and create that now inside of your views folder.

So inside your errors folder, which is inside the views folder, create a file named list.blade.php.



Then put the following code in it.

Gist:

[list.blade.php](#)

```
@if (count($errors) > 0)

<div class="alert alert-danger">

<strong>Whoops!</strong> There were some problems with your input.

<br>
<br>

<ul>

    @foreach ($errors->all() as $error)

        <li>{{ $error }}</li>

    @endforeach

</ul>
```

```
</ul>

</div>

@endif
```

So here again you can see Blade's wonderful syntax at work. We use the if statement and the count method to determine if there are errors, and if so, we use the foreach to loop over and echo each one.

So now you have this option, if you want to use it, all you would do is add the following to your view:

```
@include('layouts.errors')
```

I've moved away from this approach because I like placing the errors inline, and the out of the box views from Laravel for login and registration provide nice examples of this, so that is how we are going to handle our errors. You will see this in action shortly.

Ok, we're ready to make our create form. Let's go to create.blade.php and put the following contents in it.

Gist:

[create.blade.php](#)

From book:

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

@section('title')
    <title>Create a Widget</title>

@endsection

@section('content')

    <ol class='breadcrumb'>
        <li><a href='/'>Home</a></li>
        <li><a href='/widget'>Widgets</a></li>
        <li class='active'>Create</li>
    </ol>
```

```
<h2>Create a New Widget</h2>

<hr/>

<form class="form"
      role="form"
      method="POST"
      action="{{ url('/widget') }}>

    {{ csrf_field() }}

    <!-- name Form Input -->

<div class="form-group{{ $errors->has('name') ? ' has-error' : '' }}>

    <label class="control-label">
        Widget Name
    </label>

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

    @if ($errors->has('name'))

        <span class="help-block">
            <strong>{{ $errors->first('name') }}</strong>
        </span>

    @endif

</div>

<div class="form-group">

    <button type="submit"
            class="btn btn-primary btn-lg">
        Create
    </button>
</div>
```

```
</button>

</div>

</form>

@endsection
```

This form is similar to what we saw in the register.blade.php file, only simpler, since we only have one input field.

Obviously, we start by extending our master page. Then we add our page title:

```
@section('title')

<title>Create a Widget</title>

@endsection
```

Then we get into our content section, where we have some simple markup using Bootstrap to set our breadcrumb:

```
<ol class='breadcrumb'>
  <li><a href='/'>Home</a></li>
  <li><a href='/widget'>Widgets</a></li>
  <li class='active'>Create</li>
</ol>
```

Next let's look at how we open the form:

```
<form class="form"
      role="form"
      method="POST"
      action="{{ url('/widget') }}>

{{ csrf_field() }}
```

You can see we are setting the action to /widget, which is a route we get with our route resource. The route resource has already assigned /widget post request to the store method on the WidgetController, so all of that is done for you.

We have also included the csrf token in a hidden field for us by using the csrf_field() method. You can see just how easy this all is to work with.

Next we determine the style of the form group:

```
<div class="form-group{{ $errors->has('name') ? ' has-error' : '' }}>
```

If there is an error for name, we get the has-error style, otherwise it's a plain form group from bootstrap.

Next is the input field:

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

As I mentioned in an earlier chapter, the old() method allows you to return a value when there is an error, which is more user-friendly than having it simply disappear.

Below the input text input field, we have an if statement:

```
@if ($errors->has('name'))  
  
    <span class="help-block">  
  
        <strong>{{ $errors->first('name') }}</strong>  
  
    </span>  
  
@endif
```

So if there is an error, it will display it directly beneath the input. We are doing it this way as opposed to including the list.blade.php partial because that will simply print a list of errors all at once. Like I said before, I like the idea of giving users feedback where the error is occurring.

So that covers the interesting parts of the file. Please note if you fail to open or close your blade directives such as @endsection correctly, you get errors that are not so explicit as to the cause. So if you see a view error that you don't recognize, look for those kinds of errors.

Obviously I've given you a very granular look at this form. The reason is that you can see it will function very nicely as a template that you can copy and paste into other folders. And because we didn't complicate the form, we don't have a lot to chop out if we use it for something else.

If you are not quite sure about this, it's fine, we will see it in action later in the book and it will make a lot of sense.

Now if you point your browser to:

sample-project.com/widget/create

You should see the following:



Now if you click the Create Widget button, you get a blank screen, which is expected behavior. If you don't get a blank screen, then there is most likely a problem with your route.

Store Method

Ok, we're ready to move on to our store method. Let's go back to our WidgetController and modify the store method to the following.

Gist:

[store method](#)

From book:

```
public function store(Request $request)
{
    $this->validate($request, [
        'name' => 'required|unique:widgets|string|max:30',
    ]);

    $widget = Widget::create(['name' => $request->name]);

    $widget->save();

    alert()->success('Congrats!', 'You made a Widget');

    return Redirect::route('widget.index');
}
```

And before I go any further, I made a promise you in the beginning of the book about reminding you to include the proper use statements. The boilerplate Controller that artisan makes for us does not include a use statement for the model, in this case Widget. We also need a use statement for redirect.

So at the top of your controller, in the use statement section, add:

```
use App\Widget;
use Redirect;
```

Since we're storing to the DB via the Widget model, we need visibility on that class. We are also using the redirect method to point to the index route.

Ok, let's take a close look at our store method. You can see in the signature that we take in a request object:

```
public function store(Request $request)
{
```

Die and Dump

If we want to see what's in the request object, we can use Laravel's die and dump method, which is a combination of var_dump and die. You could use it like this:

```
public function store(Request $request)
{
    dd($request);
```

It doesn't matter what the rest of the method is, as long as you have the closing bracket because it will die after it outputs the data.

Try submitting a widget from your create form and then check out the contents of request, it will look like this, if you can imagine arrows instead of boxes:

```

Request {#40 []
  #json: null
  #convertedFiles: null
  #userResolver: Closure {#152 []}
  #routeResolver: Closure {#151 []}
  +attributes: ParameterBag {#42 []}
  +request: ParameterBag {#41 []
    #parameters: array:2 []
      "_token" => "Xg7ukx43sKIWad3FmIItXHXFL1IpKvBWLT8bBPp"
      "name" => "MyWidget"
    ]
  }
  +query: ParameterBag {#48 []}
  +server: ServerBag {#45 []}
  +files: FileBag {#44 []}
  +cookies: ParameterBag {#43 []}
  +headers: HeaderBag {#46 []}
  #content: null
  #languages: null
  #Charsets: null
  #encodings: null
  #acceptableContentTypes: null
  #pathInfo: "/widget"
  #requestUri: "/widget"
  #baseUrl: ""
  #basePath: null
  #method: "POST"
  #format: null
  #session: Store {#182 []}
  #locale: null
  #defaultLocale: "en"
}

```

Note that you have drill down into the ParameterBag to see the array holding the request values.

We get the token and the name, which in this case was named Myname from the input on the form.

The dd() method is an invaluable tool for troubleshooting bugs and determining the contents of an object. You will use it often during normal development.

Ok, so chop out the dd(\$request) out of your controller and let's move on.

Now that we have had a look at our request object, we see the type hint in the signature of the method gives us a fully instantiated Request object named \$request. It holds the form values, which we can use when we run our validation:

```
$this->validate($request, [  
    'name' => 'required|unique:widgets|string|max:30',  
]);
```

The validate method comes from the ValidatesRequests trait, which is used by the base controller that we extend with WidgetController, so we have access to it.

It takes the request instance as the first argument, and again, this is holding our values from the form, and it compares it to the rules we are handing in as the key value pairs in the array that is the second argument.

```
'name' => 'required|unique:widgets|string|max:30',
```

It sounds so complicated to describe it, but looking at it, you can see it's very easy to understand. You can see the rules are separated by a pipe. In the unique rule, we need to specify the table name.

Alpha_num means it can only be alphanumeric characters, no spaces. You can see the available rules in the docs:

Validation Rules

Finally you can see that we have a max length of 30 characters.

Next in the store method, we create the record and save it:

```
$widget = Widget::create(['name' => $request->name]);  
  
$widget->save();
```

This is where Eloquent really shines. You can see how intuitive the syntax is. We call the create method on an Eloquent model, Widget, and we assign the column 'name' to the \$request value 'name.'

Then we call the save method and we have a new widget in our DB. So we alert the user:

```
alert()->success('Congrats!', 'You made a Widget');
```

So our sweet alert implementation is going to come in handy here, again the syntax is just beautiful.

Finally, we redirect:

```
return Redirect::route('widget.index');
```

So in this case, we want to redirect to the index page for widget. We don't have that yet, but we will shortly.

Now if you didn't have the \$fillable property set on the model, as we did before to the following:

```
protected $fillable = ['name'];
```

It would cause a problem when you try to create a widget using the create form. You would get something like the following error:

```
MassAssignmentException in Model.php line 445:name
```

This will happen if we do not explicitly allow a column to be updated in the database by adding it to our fillable property on the model. I'm mentioning this because it's likely you will occasionally forget to set your fillable property and you will see this error. You can comment out the \$fillable property and try to create a widget if you want to force the error. At least now you know what it means.

Ok, once you are ready to go, and you are sure your \$fillable property is set, you can go ahead and create some widgets to try this out. You will get a blank page after each one, which is expected behavior at this point, since our index method is just an empty stub.

Index Method

This is one of those points in the book where I'm going to show you how to list your widget records on the index page, even though in reality, you probably won't use it like this. The reason for this is that even though Laravel comes with awesome tools for making a list of models, you need javascript to manipulate the data with things like column sorts and keyword searches.

In a future chapter, we are going to jump into using Elixir and Vue.js to create a robust datagrid, with column sorts and search. It will give us a much more flexible solution for displaying data.

But for now we will start with something far more basic, and take it one step at a time. Let's change your index method on your WidgetController to the following:

```
public function index()
{
    $widgets = Widget::all();

    return view('widget.index', compact('widgets'));
}
```

You can see we are using Eloquents all method to retrieve all of our widgets. It really is simple syntax.

Whenever you are curious about the Eloquent methods available to you return data from the database, consult the [docs on Eloquent](#). The docs are extremely well-written and easy to follow. We will of course be covering more methods in the course of this book as well.

Next, we need to go into our views/widget folder and create index.blade.php with the following contents:

Gist:

[index.blade.php](#)

From book:

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

@section('title')
    <title>Widgets</title>

@endsection

@section('content')
    <ol class='breadcrumb'>
        <li><a href='/'>Home</a></li>
        <li class='active'>Widgets</li>
    </ol>

    <h2>Widgets</h2>

    <hr/>

    @if($widgets->count() > 0)

        <table class="table table-hover table-bordered table-striped">

            <thead>
                <th>Id</th>
                <th>Name</th>
                <th>Date Created</th>
            </thead>

            <tbody>

                @foreach($widgets as $widget)

                    <tr>

                        <td>
                            {{ $widget->id }}
                        </td>
                        <td>
                            <a href="/widget/{{ $widget->id }}">

```

```
    {{ $widget->name }}
```

```
</a>
```

```
</td>
```

```
<td>
```

```
    {{ $widget->created_at }}
```

```
</td>
```

```
</tr>
```

```
@endforeach
```

```
</tbody>
```

```
</table>
```

```
@else
```

```
Sorry, no Widgets
```

```
@endif
```

```
@endsection
```

Assuming you have your widget data in the DB, you will see something like this:

Id	Name	Date Created
1	consequatur	2016-08-20 21:38:53
2	et	2016-08-20 21:38:53
3	ratione	2016-08-20 21:38:53
4	maxime	2016-08-20 21:38:53
5	omnis	2016-08-20 21:38:53
6	enim	2016-08-20 21:38:53
7	quas	2016-08-20 21:38:53
8	praesentium	2016-08-20 21:38:53
9	eos	2016-08-20 21:38:53
10	corrupti	2016-08-20 21:38:53
11	sequi	2016-08-20 21:38:53
12	eligendi	2016-08-20 21:38:53
13	dignissimos	2016-08-20 21:38:53
14	unde	2016-08-20 21:38:53

So let's review the view code. You can see that we wrapped the widget data in an if statement to check to see if we have any results:

```
@if($widgets->count() > 0)
```

If there are no widgets, then we execute the else statement:

```
@else
```

Sorry, no Widgets

```
@endif
```

Then we set up our table headings and our data rows. The data is inside a foreach loop to iterate over the \$widgets object and return every instance of a widget:

```

@foreach($widgets as $widget)

<tr>
  <td>{{ $widget->id }}</td>
  <td><a href="/widget/{{ $widget->id }}">{{ $widget->name }}</a></td>
  <td>{{ $widget->created_at }}</td>
</tr>

@endforeach

```

So we are using blade's {{ }} syntax to echo out the values we need.

In the case of the name, we want to turn that into a clickable link to the individual record, which is the show route. Obviously we haven't built the show view or controller method yet.

Natively, the show route has the following uri pattern:

```
/widget/{widget}
```

The parameter represented by {widget} is the id. And that will work perfectly fine, but it isn't search engine friendly. We will work on making it search engine friendly, but first, let's paginate our results. This is incredibly easy to do in Laravel.

Pagination

First, let's change the index method on our WidgetController to the following:

```

public function index()
{
    $widgets = Widget::paginate(10);

    return view('widget.index', compact('widgets'));
}

```

You can see we just altered one line, so the Widget Eloquent model uses the paginate method. The paginate method is powerful because behind the scenes it handles all the meta data need for pagination, including chunking the results and handing the current page and other data to the view.

In the widget/index.blade.php view, we simply add one line:

```
@endif
```

```
{{ $widgets->links() }}
```

I put the @endif line in there so you know where to place it. And with that, you get the following:

Id	Name	Date Created
1	ut	2017-01-25 22:40:55
2	et	2017-01-25 22:40:55
3	omnis	2017-01-25 22:40:55
4	distinctio	2017-01-25 22:40:55
5	sint	2017-01-25 22:40:55
6	magnam	2017-01-25 22:40:55
7	explicabo	2017-01-25 22:40:55
8	vero	2017-01-25 22:40:55
9	labore	2017-01-25 22:40:55
10	maiores	2017-01-25 22:40:55

Feel free to put the links in a div with a pull-right bootstrap class if you want the links to appear on the far right. It's not necessary for this demonstration.

Also note, that you can change the pagination style by going into the resources/views/vendor/pagination folder and changing the those files.

Anyway, that's full featured pagination with almost no effort. It's perfect for small datasets, if you don't need column sorts or to be able to search the data, calculate against the data, etc. But if you need something more robust, and many applications will need a more robust solution, then we have to create a data grid out of javascript. Like I mentioned earlier, we will be doing this in a later chapter with Vue.js.

Create Button

One bit of cleanup. Let's add a create button below the pagination, so we have navigation to creating a new widget:

```
{{ $widgets->links() }}
```

```
<div>
```

```
  <a href="/widget/create">
```

```
    <button type="button"
            class="btn btn-lg btn-primary">
```

```
      Create New
```

```
    </button>
  </a>
```

```
</div>
```

I put the \$widget->links in there for reference on placement. So now we have our button to create a widget.

Slugs

One thing that is typical for a show route/method/view is use of a slug to help identify the page and make it friendlier to search engines. A slug is a simple string, typically with a dash dividing the words that is based on a name, title or some other algorithm. For example, we could have a widget with the following url:

```
sample-project/widget/10-my-widget-name
```

in this case, 10-my-widget-name is the slug. I'm attaching the id to make the slug unique, which in this case isn't really necessary, since the widget name itself is unique. But in other cases, a name won't be unique, like a post title for example, and I think it's better to have the unique identifier for SEO purposes.

You won't necessarily need a slug itself for every show view, but on the other hand, if we bake it into our prototype, we have it handy when we need it.

The other thing that is typical when creating records is to record the user id of the user who is creating them. This becomes important for controlling access, so it's worth the extra work to make the modifications.

So what we need to do is rollback our migration on the widget table, make a few adjustments to everything and then move on. We can rollback the widget migration by using:

```
php artisan migrate:rollback
```

Alternatively, we could have made a new migration to modify the widgets table, but since the last migration we did was to create the widgets table, it's simpler to do it this way. So let's do that now.

Once that is done, let's go back to our migration file and modify the up method as follows:

Gist:

[widget migration](#)

From book:

```
public function up()
{
    Schema::create('widgets', function (Blueprint $table) {

        $table->increments('id');
        $table->integer('user_id')->unsigned();
        $table->string('name')->unique();
        $table->string('slug')->unique();
        $table->timestamps();

    });
}
```

So we added a user_id column that is an integer, which must be unsigned. The unsigned constraint forces it to be a positive number and matches the data type of the record it will point to, which is the id column, also unsigned.

We could add a foreign key, but we don't need it here. Some developers like using foreign keys, some don't. Most of the developers I know avoid foreign keys because of the way it complicates development. So I tend to follow that direction.

For convenience, I also changed the column type on 'name' to 'string', which allows us to put spaces in the name, which we will use to demonstrate the slugs.

Moving on, we also have the slug column with a unique constraint on the column to finish up. So go ahead and run:

```
php artisan migrate
```

Neither of these column additions require a change to the create form view. However, we do have quite a few other changes to make.

Changing the Store Method

In our validate method, inside the store method on our WidgetController, we need to change the alpha_num to string like so:

```
$this->validate($request, [  
    'name' => 'required|unique:widgets|string|max:30',  
]);
```

Create the Slug

Laravel ships with a very handy helper method that is available globally called str_slug(). The first parameter it takes is the string to be parsed, and the second is the string to insert in the spaces.

```
$slug = str_slug($request->name, "-");
```

You can see we are handing in the widget ‘name’ via the \$request object. If there are no spaces, it still returns a value, so this works perfectly for slug creation.

We need to add this line to our store method in between the validation and create method calls.

Changing the Create Method

We need to add our create method the new values we want saved:

```
$widget = Widget::create(['name' => $request->name,
                         'slug' => $slug,
                         'user_id' => Auth::id()]);
```

You can see we are using Auth::id() to set the id of the currently logged in user.

Here is the gist for the method if you need it:

[WidgetController store method](#)

Add Auth Use Statement

In order to use Auth::id(), we must add to the use statement to the WidgetController:

```
use Illuminate\Support\Facades\Auth;
```

Note that in the views, Auth::id() is available to us without needing a use statement.

Just for reference, I'm going to give you a gist of the WidgetController as it stands at this point. That way you can check your code against it if you have any problems:

[WidgetController](#)

Change \$fillable Property on Widget Model

Since we're adding new columns in the DB, we need to update our model to allow these columns in mass assignment:

```
protected $fillable = ['name',
                      'slug',
                      'user_id'];
```

Basic Relationships

Relationships will get more coverage later on, we're only going to touch on here, since we are in workflow and it's good practice to set the relationship.

Laravel's syntax for relationships is incredibly easy:

On Widget.php, add the following method:

```
/*
 * Get the user that owns the widget.
 */
public function user()
{
    return $this->belongsTo('App\User');
}
```

Many widgets can belong to a user, so we use belongsTo as the relationship type.

On User.php, add the following method:

```
public function widgets()
{
    return $this->hasMany('App\Widget');
}
```

One user can have many widgets, so we use hasMany.

So in both cases, Laravel looks for the user_id on Widget and matches it to id on User. This allows for the following:

```
$widgets = App\User::find(1)->widgets()->where('name', 'foo')->first();
```

In the above example, we can chain the widgets method from the User model, which defines the relationship to pull in the widget that belongs to the user.

You can see the where method is intuitive, essentially, give me the widget where the name is equal to foo that belongs to user id 1.

Because of the relationship, the User model knows which widgets belong to it. It makes it very easy to work with. This isn't really much of a practical example here, but relationships will be explored in more detail at a later point in the book.

Tinker comes in handy for testing relationships. Let's give it a try. First, make sure you are logged in.

Now use the widget create form to make a few widgets, then call up tinker from the command line:

```
php artisan tinker
```

Then run:

```
$widgets = App\User::find(1)->widgets()->get();
```

We're telling it to use the find method to find the user with an id of 1, then get that user's associated widgets. Assuming you have a user id of 1 and some widgets created under that user, the result will look like this:

```
>>> $widgets = App\User::find(1)->widgets()->get();
=> Illuminate\Database\Eloquent\Collection {#706
    all: [
        App\Widget {#707
            id: 2,
            user_id: 1,
            name: "incidunt recusandae",
            slug: "incidunt-recusandae",
            created_at: "2016-08-21 04:05:50",
            updated_at: "2016-08-21 04:05:50",
        },
        App\Widget {#708
            id: 5,
            user_id: 1,
            name: "illum similiq",
            slug: "illum-similiq",
            created_at: "2016-08-21 04:05:50",
            updated_at: "2016-08-21 04:05:50",
        },
        App\Widget {#709
            id: 7,
            user_id: 1,
            name: "vel amet",
            slug: "vel-amet",
            created_at: "2016-08-21 04:05:50",
            updated_at: "2016-08-21 04:05:50",
        },
        App\Widget {#710
            id: 10,
            user_id: 1,
            name: "quia non",
            slug: "quia-non",
            created_at: "2016-08-21 04:05:50",
            updated_at: "2016-08-21 04:05:50",
        },
        App\Widget {#711
            id: 11,
            user_id: 1,
            name: "necessitatibus quo",
            slug: "necessitatibus-quo",
        },
    ],
}
```

So now you have a quick way to test if your relationship is working, without having to use the test controller, without having to write code that you are not going to use.

This is yet another example of why Laravel is by far the best PHP framework out there.

Add Middleware to WidgetController

Now we get a brief introduction into middleware. The reason for that is we are modifying creating a widget, and in order to create a widget, we now must require the user be logged in. It's the only way we can know their user_id when we save the record.

Laravel comes with ready-made middleware that will enforce the authentication requirement. There are multiple ways to use it, and we will talk more about that later, but the primary way I find myself using it is through the constructor on the controller.

The boilerplate controllers don't come with constructors, so you have to add it. On the WidgetController, add the following:

```
public function __construct()
{
    $this->middleware('auth', ['except' => ['index', 'show']]);
}
```

For reference:

[Full WidgetController.php](#)

We are requiring the 'auth' middleware on all methods, except index and show. Typically, but not always, index and show will be visible to guests. If we want to require login on everything, we would simply remove that parameter entirely like so:

```
public function __construct()
{
    $this->middleware('auth');
}
```

But that's not what we want now, so make sure you have the exceptions.

Our middleware is making a call to Authenticate.php, which is in vendor/laravel/framework/src/Illuminate/Auth/Middleware folder. The authenticate method is as follows:

```
protected function authenticate(array $guards)
{
    if (empty($guards)) {

        return $this->auth->authenticate();
    }

    foreach ($guards as $guard) {

        if ($this->auth->guard($guard)->check()) {

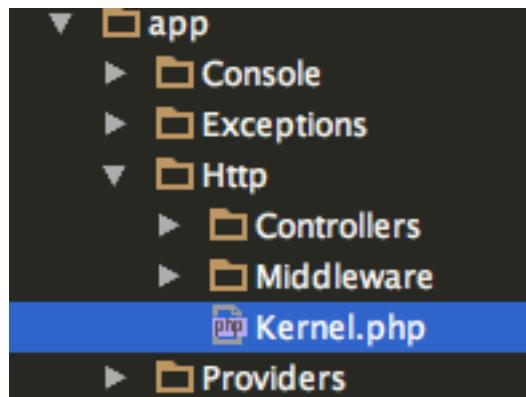
            return $this->auth->shouldUse($guard);

        }
    }

    throw new AuthenticationException('Unauthenticated.', $guards);
}
```

You can see it's checking against multiple guards. I don't want to go too deep here, but I did want to point out that Authenticate.php is residing in the framework, so we will not be modifying this at all.

Now if you are wondering why the class is called Authenticate and yet the middleware is looking for 'auth', that gets set in the Kernel.php file, which is in your Http folder:



In the protected \$routeMiddleware array, we see 'auth' points to the Authenticate class:

```
protected $routeMiddleware = [  
  
    'auth' => \Illuminate\Auth\Middleware\Authenticate::class,  
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,  
    'bindings' => \Illuminate\Routing\Middleware\SubstituteBindings::class,  
    'can' => \Illuminate\Auth\Middleware\Authorize::class,  
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,  
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,  
  
];
```

Later, when we create our own middleware, we will need to alias it here.

Ok, that was a brief introduction to middleware in the middle of workflow to a fairly big topic. I know it's a lot to digest, so don't worry about memorizing it. We will be returning to it later. For now, all you need to know is that users have to be logged in before they can create a widget.

Change Routes to Widget

We need to make a change to our routes by modifying the resource and adding routes for show and create in our routes/web.php file.

```
// Widget routes

Route::get('widget/create', 'WidgetController@create')->name('widget.create');

Route::get('widget/{id}-{slug?}', 'WidgetController@show')->name('widget.show');

Route::resource('widget', 'WidgetController', ['except' => ['show', 'create']]);

```

In the route resource, we hand in a second parameter that tells the resource not to set the show route or the create route. We need to define those separately.

The reason why we have defined create is that the new show route is a pattern match for create and will take precedence over it, unless we do that. So we define create.

Then comes the route for show. You can see the following:

```
{id}-{slug?}
```

The question mark on the slug means it's optional, so we are setting this up so that the controller method will work with or without the slug.

Note: We do not use \$ in the route parameters. That is a common error, so try to avoid doing that.

Notice we are naming the route with the name method:

```
->name('widget.show');
```

A route resource will automatically name the route, but when you are declaring the route individually, you have to name the route yourself. Now actually you don't have to name it, it's just good practice because it makes it easier to work with.

Also note the placement of the create and show routes are above the route resource and this is important for precedence. Sometimes the routes will get confused if they are the other way around because Laravel might assume the route parameters are already taken.

Modify index view

In our views/widget/index.blade.php file, we need to change the table row for the link to:

```
<td>

<a href="/widget/{{ $widget->id }}-{{ $widget->slug }}">
{{ $widget->name }}</a>

</td>
```

You can see that we are matching the pattern we created in our route file.

Ok, so now if you create a widget with multiple words in it, you should get redirected to the index page, which has our index grid. If you mouseover the name link, you should see the slug in the url.

If you name your widget ‘boom goes the night,’ on mouseover, you get:

```
sample-project.com/widget/1-boom-goes-the-night
```

Ok, before we move on to making our show controller method and view, let’s modify our database/factories/-ModelFactory file, so we can repopulate our seed data.

New Factory method

Gist:

[factory method](#)

From book:

```
$factory->define(App\Widget::class, function ($faker) {

    $name = $faker->unique()->word . ' ' . $faker->unique()->word;
    $slug = str_slug($name, "-");
    $user_id = rand(1,4);

    return [
        'name' => $name,
        'slug' => $slug,
        'user_id' => $user_id,
    ];
});
```

You can see we are concatenating two unique faker words separated by a space to create our name. Then we use the slug method to format the slug. And finally, we assign user_id a random number from 1 - 4 to simulate users in the system.

So now if you run php artisan tinker from the command line, then:

```
\App\Widget::truncate();
```

That will clear the table. Then run:

```
factory('App\Widget', 30)->create();
```

That will get you 30 fresh records. Then control d to quit tinker.

Show Method

Ok, now we are ready to move on to the show method on the WidgetController. Let's change that to the following:

Gist:

[Show Method](#)

From book:

```
public function show($id, $slug = '')  
{  
    $widget = Widget::findOrFail($id);  
  
    if ($widget->slug !== $slug) {  
  
        return Redirect::route('widget.show', [  
            'id' => $widget->id,  
            'slug' => $widget->slug  
        ], 301);  
  
    }  
  
    return view('widget.show', compact('widget'));  
}
```

Ok, so you can see in the signature, we take in the record id and a slug that defaults to an empty string. Remember that we made it optional on the route, and defaulting it to an empty string allows us to bring it into the method, even if it hasn't been provided.

Next we do a lookup of the record we want:

```
$widget = Widget::findOrFail($id);
```

That's a really beautiful part of eloquent. If it doesn't find a matching record, it returns a `NotFoundHttpException`, which you can then setup to handle to return a view.

Right now, if you type in the following:

```
http://sample-project.com/djhdfjfhdf
```

You get:

Sorry, the page you are looking for could not be found.

1/1 [NotFoundException in RouteCollection.php line 161:](#)

```
1. in RouteCollection.php line 161
2. at RouteCollection->match(object(Request)) in Router.php line 746
3. at Router->findRoute(object(Request)) in Router.php line 655
4. at Router->dispatchToRoute(object(Request)) in Router.php line 631
5. at Router->dispatch(object(Request)) in Kernel.php line 236
6. at Kernel->Illuminate\Foundation\Http\{closure}(object(Request))
7. at call_user_func(object(Closure), object(Request)) in Pipeline.php line 139
8. at Pipeline->Illuminate\Pipeline\{closure}(object(Request)) in VerifyCsrfToken.php line 50
9. at VerifyCsrfToken->handle(object(Request), object(Closure))
10. at call_user_func_array(array(object(VerifyCsrfToken), 'handle'), array(object(Request), object(Closure))) in Pipeline.php line 124
11. at Pipeline->Illuminate\Pipeline\{closure}(object(Request)) in ShareErrorsFromSession.php line 49
12. at ShareErrorsFromSession->handle(object(Request), object(Closure))
13. at call_user_func_array(array(object(ShareErrorsFromSession), 'handle'), array(object(Request), object(Closure))) in Pipeline.php line 124
14. at Pipeline->Illuminate\Pipeline\{closure}(object(Request)) in StartSession.php line 62
15. at StartSession->handle(object(Request), object(Closure))
16. at call_user_func_array(array(object(StartSession), 'handle'), array(object(Request), object(Closure))) in Pipeline.php line 124
17. at Pipeline->Illuminate\Pipeline\{closure}(object(Request)) in AddQueuedCookiesToResponse.php line 37
18. at AddQueuedCookiesToResponse->handle(object(Request), object(Closure))
19. at call_user_func_array(array(object(AddQueuedCookiesToResponse), 'handle'), array(object(Request), object(Closure))) in Pipeline.php line 124
20. at Pipeline->Illuminate\Pipeline\{closure}(object(Request)) in EncryptCookies.php line 59
21. at EncryptCookies->handle(object(Request), object(Closure))
22. at call_user_func_array(array(object(EncryptCookies), 'handle'), array(object(Request), object(Closure))) in Pipeline.php line 124
23. at Pipeline->Illuminate\Pipeline\{closure}(object(Request)) in CheckForMaintenanceMode.php line 42
24. at CheckForMaintenanceMode->handle(object(Request), object(Closure))
25. at call_user_func_array(array(object(CheckForMaintenanceMode), 'handle'), array(object(Request), object(Closure))) in Pipeline.php line 124
26. at Pipeline->Illuminate\Pipeline\{closure}(object(Request))
27. at call_user_func(object(Closure), object(Request)) in Pipeline.php line 103
28. at Pipeline->then(object(Closure)) in Kernel.php line 122
29. at Kernel->sendRequestThroughRouter(object(Request)) in Kernel.php line 87
30. at Kernel->handle(object(Request)) in index.php line 54
```

You get all the debug info because we are in debug mode, you would just get the top message if you were in production. But that is still not what we want. It would be much better to show the user a message within the context of our site.

So we will want to set this up, but let's not do it at the moment since we are in the middle of working through our show method, we'll get back to it before the end of the chapter.

Next in the show method, we check to see if the slug handed in matches the one on the record:

```
if ($widget->slug !== $slug) {  
  
    return Redirect::route('widget.show', [  
  
        'id' => $widget->id,  
        'slug' => $widget->slug  
  
, 301);  
  
}
```

If it matches we go on, but if not, we redirect with back to the route with the slug provided this time, so we can make sure the route is appended to the url. The 301 parameter is for the header to let search engines know it's a permanent redirect.

Once we're good there, we redirect to the view:

```
return view('widget.show', compact('widget'));
```

So we got the view and pass the widget object along with it.

Now we're ready to create our view file. Let's make a show.blade.php within our widget view folder and place the following within it:

Gist:

[show.blade.php](#)

From book:

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

@section('title')
    <title>{{ $widget->name }} Widget</title>

@endsection

@section('content')
    <ol class='breadcrumb'>
        <li><a href='/'>Home</a></li>
        <li><a href='/widget'>Widgets</a></li>
        <li><a href='/widget/{{ $widget->id }}'>
            {{ $widget->name }}</a>
        </li>
    </ol>

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

    <hr/>

    <div class="panel panel-default">

        <!-- Table -->
        <table class="table table-striped">
            <thead>
                <tr>

                    <th>Id</th>
                    <th>Name</th>
                    <th>Date Created</th>
                    <th>Edit</th>
                    <th>Delete</th>

                </tr>
            </thead>

            <tbody>
                <tr>
                    <td>

                        {{ $widget->id }}

                    </td>
                </tr>
            </tbody>
        </table>
    </div>
```

```
</td>
<td>

    <a href="/widget/{{ $widget->id }}/edit">
        {{ $widget->name }}
    </a>

    </td>
    <td>

        {{ $widget->created_at }}

    </td>
    <td>

        <a href="/widget/{{ $widget->id }}/edit">
            <button type="button"
                    class="btn btn-default">
                Edit
            </button></a>
        </td>

        <td>

            <div class="form-group">

                <form class="form"
                      role="form"
                      method="POST"
                      action="{{ url('/widget/'. $widget->id) }}>

                    <input type="hidden"
                           name="_method"
                           value="delete">

                    {{ csrf_field() }}

                    <input class="btn btn-danger"
```

```
        Onclick="return ConfirmDelete();"
        type="submit"
        value="Delete">

    </form>
</div>
</td>

</tr>
</tbody>

</table>

</div>

@endsection

@section('scripts')

<script>

    function ConfirmDelete()
    {
        var x = confirm("Are you sure you want to delete?");

        return x;
    }

</script>

@endsection
```

Please use the gist for better formatting of the code if you are copying directly.

Most of this we have seen before, so we don't have to be quite so granular in our analysis.

We're using a table for our individual widget record. The last column has a delete action. You could easily use something other than a table, I just used it for consistency.

We want delete to be triggered by a post method to protect against someone just coming along and deleting our records via a GET request. So we put a form in the last table row.

You'll note that we had to add the following:

```
<input type="hidden"  
      name="_method"  
      value="delete">
```

We do this because we need to identify the right method to Laravel and html only supports get and post for form requests.

So this line is a way of spoofing the form to make sure Laravel understands the request properly.

In the bottom, in the scripts section, we have a small amount of javascript to put a confirmation button on delete, so that the user doesn't inadvertently delete records.

Ok, so if you have everything correct, it should look something like this:

Id	Name	Date Created	Edit	Delete
4	eveniet pariatur	2017-01-26 02:44:15	Edit	Delete

© 2015 - 2017 Sample Project All rights Reserved.

Right away you can see that we have a date format that isn't very friendly. In most cases you will not be displaying the minutes and hours in show record, although you could, it is certainly up to you.

Accessors and Mutators

Laravel has some really cool formatting features available to via the Eloquent Model. They are Accessors and Mutators, which is what we used to call getters and setters. An Accessor always calls the attribute a certain way and a Mutator always saves the attribute a certain way.

We can solve our date format issue with a simple accessor method on the Widget model class. Let's add the following to Widget.php:

```
public function getCreatedAtAttribute($value)
{
    return Carbon::parse($value)->format('m-d-Y');
}
```

You can see the convention is to take the created_at attribute and name the method getCreatedAtAttribute. It takes in the value you want to operate on, then in our case, we are using Carbon to reformat the date.

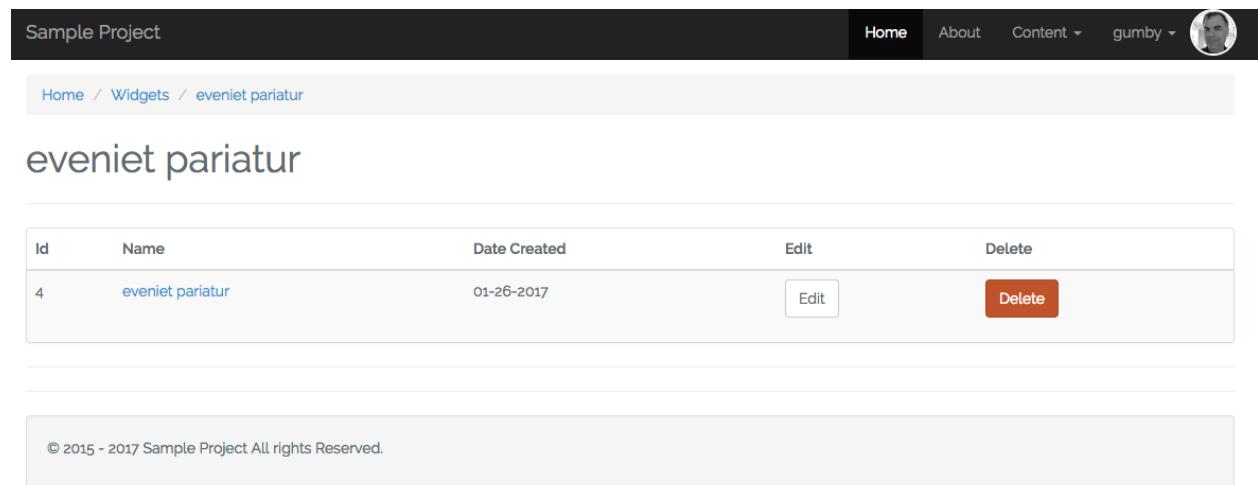
Laravel uses Carbon and it's just another excellent choice because Carbon, authored by Brian Nesbitt, is easy to work with and well-documented. When you have time, check out [Carbon](#).

To use Carbon, you have to include in the use statements at the top of the Widget.php file:

```
use Carbon\Carbon;
```

So once you have done that, your date format is automatically updated for wherever we are accessing the created_at date, which we use in our views.

Now the show record looks like this:



Id	Name	Date Created	Edit	Delete
4	eveniet pariatur	01-26-2017	Edit	Delete

© 2015 - 2017 Sample Project All rights Reserved.

It's worth talking about some alternatives to this approach. One possibility, a bad one, is that you could modify the framework model class, which the models are extending. I really avoid this at all costs. The model class has over 3500 lines of code to it. If you change it, and then for some reason you have to run composer update, you could potentially overwrite your code.

There may be cases where that's unavoidable, and depending on your level of expertise as a programmer, you may be more comfortable going that route, so there are no absolutes here, just saying I would personally never do it.

A more common approach is to extend the model class to a new class and then extend your models from that. If you do it this way, you don't change any framework code, but you still get the customization to the model that you want by using simple inheritance.

Within your app folder, let's create the following file:

SuperModel.php

From book:

```
<?php
namespace App;

use Illuminate\Database\Eloquent\Model;
use Carbon\Carbon;

class SuperModel extends Model
{

    public function getCreatedAtAttribute($value)
    {

        return Carbon::parse($value)->format('m-d-Y');
    }
}
```

So now let's change the Widget.php to:

Gist:

[Widget Revised](#)

From book:

```
<?php

namespace App;

class Widget extends SuperModel
{
    protected $fillable = ['name',
                          'slug',
                          'user_id'];

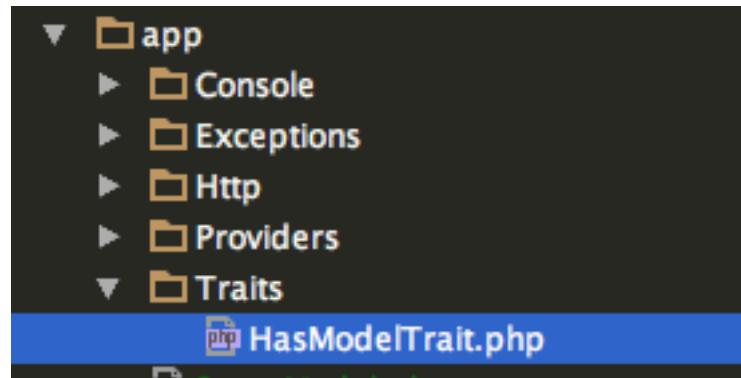
    /**
     * Get the user that owns the widget.
     */

    public function user()
    {
        return $this->belongsTo('App\User');
    }
}
```

So now if you check it, we have the date formatted as we did before, but now any model that extends SuperModel will automatically inherit the accessor. That means you can still generate a model with artisan like before, you just have to remember to change it to extend SuperModel instead of Model if you want access to the SuperModel methods.

A third way we can extend methods to all models is to create a trait. This is a little more defensive in that you have to explicitly bring in the methods of a trait to each model that you want to use it on.

Let's demonstrate by creating a folder in our app folder named Traits, and within that a file named HasModelTrait.php:



So let's add to HasModelTrait.php the following contents:

Gist:

[HasModelTrait](#)

From book:

```
<?php

namespace App\ Traits;

use Carbon\Carbon;

trait HasModelTrait
{

    public function getCreatedAtAttribute($value)
    {

        return Carbon::parse($value) -> format('m-d-Y');

    }

}
```

When you are creating a class or trait, pay careful attention to the namespacing:

```
namespace App\_traits;
```

I know I mentioned this before, but it will trip you up if you don't follow the convention, the autoloader won't see it. The namespace is the path to the folder in which the class or trait resides in most cases.

When you use the trait in another class, the use statement looks like this:

```
use App\traits\HasModelTrait;
```

By the way, even though the app folder is lowercase, the namespace is uppercase. That can drive you nuts if you get that wrong.

Anyway, let's add the use statement to our Widget model. I'm going to show you the whole Widget.php file for reference:

```
<?php

namespace App;

use App\traits\HasModelTrait;

class Widget extends SuperModel
{
    use HasModelTrait;

    protected $fillable = [
        'name',
        'slug',
        'user_id'];

    /**
     * Get the user that owns the widget.
     */

    public function user()
```

```
{  
  
    return $this->belongsTo('App\User');  
  
}  
}
```

So not only did we add the use statement under the namespace, we also had to put a different use statement as the first line inside the class:

```
class Widget extends SuperModel  
{  
    use HasModelTrait;
```

I'm being very explicit on setting up a trait because I want you to be comfortable with traits. They are a very useful programming pattern and I'm seeing them used more and more. I like them a lot myself.

That said, you may have noticed that we are still extending from SuperModel. That is because we are going back to that approach for extending common methods. So if you want to temporarily test this, chop the accessor method out of the SuperModel class, so it's an empty stub, and you will see that you still have the date formatted correctly. That means it's now pulling it in from the trait.

Ok, so since we're using extend SuperModel approach, go ahead and return those files to their prior state. I will list the gists for reference:

[Widget.php](#)

[SuperModel.php](#)

Edit Method

Ok, so we're ready to move on to the edit method on the WidgetController. This is a fairly simple method.

Gist:

[edit method](#)

From book:

```
public function edit($id)
{
    $widget = Widget::findOrFail($id);

    return view('widget.edit', compact('widget'));
}
```

We take in the id, find the record and return the model to the view. We don't have a view of course, so let's make that now.

Create in your widget view folder edit.blade.php file with the following.

Gist:

[edit form](#)

From book:

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

@section('title')
    <title>Edit Widget</title>
@endsection

@section('content')

    <ol class='breadcrumb'>
        <li><a href='/'>Home</a></li>
        <li><a href='/widget'>Widgets</a></li>
        <li><a href='/widget/{$widget->id}'>
            {$widget->name}</a></li>
        <li class='active'>Edit</li>
    </ol>

    <h1>Edit Widget</h1>
```

```
<hr/>

<form class="form"
      role="form"
      method="POST"
      action="{{ url('/widget/'. $widget->id) }}>
    {{ method_field('PATCH') }}

    {{ csrf_field() }}

    <!-- name Form Input -->

<div class="

    form-group{{ $errors->has('name') ? ' has-error' : '' }}>

        <label class="control-label">Widget Name</label>

        <input type="text"
              class="form-control"
              name="name"
              value="{{ $widget->name }}>

        @if ($errors->has('name'))

            <span class="help-block">

                <strong>{{ $errors->first('name') }}</strong>

            </span>

        @endif

    </div>

    <div class="form-group">

        <button type="submit"
               class="btn btn-primary btn-lg">
```

```
Edit  
</button>  
</div>  
</form>  
  
@endsection
```

Please make sure to use the Gist for formatting.

Since we have the model instance available to us in the view, we will use it to pass along the id like so:

```
<form class="form"  
      role="form"  
      method="POST"  
      action="{{ url('/widget/'. $widget->id) }}>
```

That's only two lines to avoid wordwrap.

We use the url method and attach the id of the record, in this case \$widget->id.

We are doing the method spoofing with the method_field helper method:

```
{{ method_field('PATCH') }}
```

We use a 'patch' request for updates.

We also assign the value of the existing name of the widget like so:

```
<input type="text" class="form-control" name="name" value="{{ $widget->name }}>
```

The rest is mostly what we have seen before. It's pretty easy to work with.

Update Method

So now we're on to the update method on the WidgetController. Let's change the update method to the following.

Gist:

[update method](#)

From book:

```
public function update(Request $request, $id)
{
    $this->validate($request, [
        'name' => 'required|string|max:30|unique:widgets,name,' . $id
    ]);

    $widget = Widget::findOrFail($id);

    $slug = str_slug($request->name, "-");

    $widget->update([
        'name' => $request->name,
        'slug' => $slug,
        'user_id' => Auth::id()]);
}

alert()->success('Congrats!', 'You updated a widget');

return Redirect::route('widget.show', [
    'widget' => $widget, 'slug' => $slug
]);
}
```

Notice on the validate method that I put the unique rule last. Like in the store method, we identify the table name, widgets, but we also identify the column, 'name.' Also note there is no space between the tablename and the column name.

Then we concatenate \$id on the end:

```
'name' => 'required|string|max:30|unique:widgets,name,' . $id
```

If we didn't do that, the validation would enforce the unique rule and not allow us to use the same name. That's normally used when you have multiple columns that can be updated and you want to keep one of the columns that is supposed to be uniques with the same value.

Now you may be asking yourself, if we only have one attribute and we are editing it, why do we have to worry about that?

Actually, we don't. I'm setting it up here this way because in almost every scenario you will have models with more than one attribute. And in those cases, where you have a name or title that must be unique, this becomes a problem.

For example, if you had a category id on this model and you wanted to change that, but you didn't want to change the widget name, you would need the above validation syntax or it would complain and fail. Coding it this way gives us a working example to refer to. I tend to trip over this myself, so I like having an example handy.

If this isn't clear now, don't worry about it, we will see it in action later.

After validating, we find the \$widget instance we want and use the Eloquent update method on it. the last thing to note is that we are passing the \$widget object into the route parameters and Laravel knows how to extract the id from it. We also pass along the slug in this case and you can see how we do that:

```
return Redirect::route('widget.show', [
    'widget' => $widget, 'slug' => $slug
]);
```

Also note that the first parameter in the route method is the name of the route. This is why we named our routes. It's a fairly common scenario to have to pass along variables and the route method, which takes in the route name, makes this easy.

This should all be working perfectly at this point. We built-in a link to update on the show page, it's the widget name in the table row. Click on that and it will take you to the edit form.

Destroy Method

In this case, we only need to create the destroy method, there is no corresponding view. So in your WidgetController, change the destroy method to the following.

Gist:

[destroy method](#)

From book:

```
public function destroy($id)
{
    Widget::destroy($id);

    alert()->overlay('Attention!', 'You deleted a widget', 'error');

    return Redirect::route('widget.index');

}
```

So this is a fairly obvious method. We hand in the id and Eloquent provides us with a simple method to delete a record.

Laravel also provides for soft deletes, but we're not using them in our sample project, so for the sake of time, I'm not going to cover them. The [docs](#) are pretty clear on how to use them though, if you want to check that out.

In our destroy method, when we're done, we alert and redirect to index.

Automatic Route Model Binding

Laravel 5.4 comes with a really cool feature known as automatic route model binding. It's sounds complicated, but it's not. Let's look at an alternative way to do our update method:

Gist:

[update method](#)

From book:

```

public function update(Request $request, Widget $widget)
{
    $this->validate($request, [
        'name' => 'required|string|max:40|unique:widgets,name,' .
        '$widget->id
    ]);

    $slug = str_slug($request->name, "-");

    $widget->update(['name' => $request->name,
                    'slug' => $slug,
                    'user_id' => Auth::id()]);
}

alert()->success('Congrats!', 'You updated a widget');

return Redirect::route('widget.show', [
    'widget' => $widget, 'slug' =>$slug
]);
}

```

Ok, so you can see in the signature, instead of bringing in \$id, we get an instance of Widget. Laravel will automatically instantiate the model record we need as long as the instance variable is named the same as the identifier on the route.

Since we used a route resource for the update method, we don't see the individual route, but it is the same as :

```
Route::patch('widget/{widget}', 'WidgetController@update');
```

You can check that by running:

```
php artisan route:list
```

Ok, getting back to the changes in the method. So we bring in an instance of Widget, which means on the validate method, instead of \$id, we get \$widget->id.

And since we already have our model instance, we have removed the call to the model:

```
$widget = Widget::findOrFail($id);
```

Note that in the case of the show method, the model binding doesn't work because we have a more complicated route to account for the slug:

```
Route::get('widget/{id}-{slug?}',  
          'WidgetController@show')  
    ->name('widget.show');
```

We can fix this by changing the route to:

```
Route::get('widget/{widget}-{slug?}',  
          'WidgetController@show')  
    ->name('widget.show');
```

It just has to match the instance variable name. And now we can change our show method.

Gist:

[show method](#)

From book:

```

public function show(Widget $widget, $slug = '')
{
    if ($widget->slug !== $slug) {

        return Redirect::route('widget.show', [
            'id' => $widget->id,
            'slug' => $widget->slug
        ], 301);
    }

    return view('widget.show', compact('widget'));
}

```

And obviously, we can rewrite the edit method:

Gist:

[edit method](#)

From book:

```

public function edit(Widget $widget)
{
    return view('widget.edit', compact('widget'));
}

```

Overall, I really like route model binding, it cuts down on the amount of code in the controller and makes for cleaner code. It is up to you which way you prefer to do it.

I should note that for the remainder of the book, I used the older way of doing it, since this is new to me as well. I will probably use route model binding instead in the next edition of the book.

Meanwhile, you can decide for yourself if you want to use route model binding in the subsequent controllers we create in the book. If you do use route model binding, just be careful to make sure the identifier on the route is the same name as the instance variable.

Error Handling

I mentioned before about what happens when you type in something like the following:

```
http://sample-project.com/kdsjfkjd
```

If you try that, you get an error message, but not within the context of your site. It would be a lot nicer if we had a formatted error page that we use instead. So let's finish this chapter by putting that together.

404.blade.php

Let's start with the view. In your views/errors folder, create a file named 404.blade.php and put the following in it.

Gist:

[404.blade.php](#)

From book:

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

@section('content')

<div class="alert alert-danger alert-dismissible alert-important"
     role="alert">

<button type="button"
        class="close"
        data-dismiss="alert"
        aria-label="Close">

<span aria-hidden="true">&times;</span></button>

<strong>Oh Snap!</strong> We can't find what you are looking for...

</div>

@endsection
```

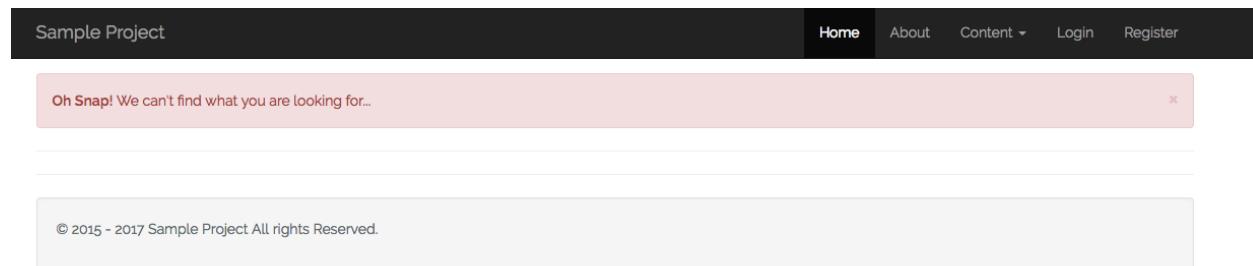
This will give us a nicely formatted error page, after we set up our handler for the exceptions.

Laravel handles both ModelNotFoundException and NotFoundHttpException automatically for you, if you have the 404.blade.php file in the errors folder.

So now you can type in some gibberish into the url:

`http://sample-project.com/kdsjfkjd`

And you will get:



Note: When responding to the 404 error, laravel does not start the session and as a result, you do not see the username and Gravatar, even if you are logged in. If you try to go to the login or register links when logged in, the RedirectIfAuthenticated middleware will send you to the '\home' uri, which of course we no longer have.

So let's change the handle method in app/Http/Middleware/RedirectIfAuthenticated.php:

```
public function handle($request, Closure $next, $guard = null)
{
    if (Auth::guard($guard)->check()) {
        return redirect('/');
    }
    return $next($request);
}
```

I'm not providing a gist because you just need to change '/home' to '/' in the redirect method.

If you want to test a ModelNotFoundException, go to your TestController and change the index method to the following:

```
public function index()
{
    $result = Widget::findOrFail(200);

}
```

Assuming that you do not have 200 records in your widget table, this will throw a ModelNotFoundException because of the findOrFail method. Laravel will also return the 404.blade.php file if it exists for this error, so let's try it.

Go to:

<http://sample-project.com/test>

You will see our nicely formatted error alert, and this time you will see it preserves the logged in state if you are logged in.

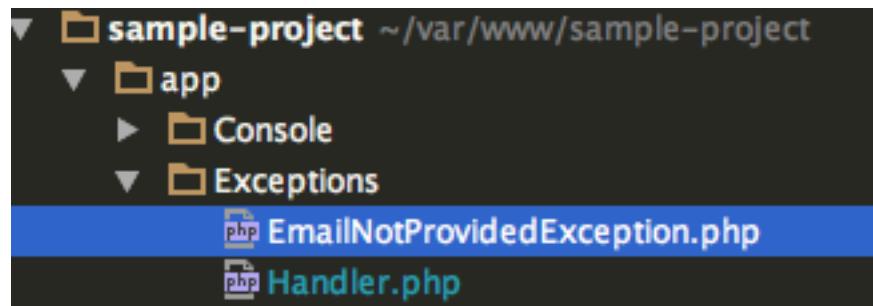
Exceptions

While the two exceptions that Laravel handles for us automatically are convenient, we do need to learn how to write custom exceptions because we need them for a variety of reasons.

Debugging and error logging are among those reasons. Many developers pipe their errors to the 3rd party applications such as [ErrorStream](#) and other companies, so that they can keep track of exceptions that are being thrown as their application is being used.

I'm not covering integrating to reporting services, but we will cover creation of custom exceptions, and later, when we build our Socialite implementation, you will see these in use and get a better understanding of how they are useful.

So let's start by creating the EmailNotProvidedException.php in the Exceptions folder:



Gist:

[EmailNotProvidedException](https://gist.github.com/evercode1/b0e889c16205baabcfcaabe362f60f0f)

From book:

```
<?php
```

```
namespace App\Exceptions;

class EmailNotProvidedException extends \Exception
{
```

You can see there isn't much to it. We give it the namespace. Then we extend PHP's Exception class. Basically all we are doing is giving a name to this particular exception, so we can track and handle it individually.

So now that we have a custom exception, we need to handle it. What we're going to do is modify our handler class inside of our app/Exceptions folder.

This is what you get out of the box:

```
<?php

namespace App\Exceptions;

use Exception;
use Illuminate\Auth\AuthenticationException;
use Illuminate\Foundation\Exceptions\Handler as ExceptionHandler;

class Handler extends ExceptionHandler
{
    /**
     * A list of the exception types that should not be reported.
     *
     * @var array
     */
    protected $dontReport = [
        \Illuminate\Auth\AuthenticationException::class,
        \Illuminate\Auth\Access\AuthorizationException::class,
        \Symfony\Component\HttpKernel\Exception\HttpException::class,
        \Illuminate\Database\Eloquent\ModelNotFoundException::class,
        \Illuminate\Session\TokenMismatchException::class,
        \Illuminate\Validation\ValidationException::class,
    ];

    /**
     * Report or log an exception.
     *
     * This is a great spot to send exceptions to Sentry, Bugsnag, etc.
     *
     * @param  \Exception  $exception
     * @return void
     */
    public function report(Exception $exception)
    {
        parent::report($exception);
    }

    /**
     * Render an exception into an HTTP response.
     *

```

```
* @param \Illuminate\Http\Request $request
* @param \Exception $exception
* @return \Illuminate\Http\Response
*/
public function render($request, Exception $exception)
{
    return parent::render($request, $exception);
}

/**
 * Convert an authentication exception into an unauthenticated response.
 *
 * @param \Illuminate\Http\Request $request
 * @param \Illuminate\Auth\AuthenticationException $exception
 * @return \Illuminate\Http\Response
*/
protected function unauthenticated
    ($request, AuthenticationException $exception)
{
    if ($request->expectsJson()) {

        return response()->json(['error' => 'Unauthenticated.'], 401);
    }

    return redirect()->guest('login');
}
```

Now let's modify that to the following:

Gist:

[Handler.php](#)

From book:

```
<?php

namespace App\Exceptions;

use Exception;
use Illuminate\Auth\AuthenticationException;
use Illuminate\Foundation\Exceptions\Handler as ExceptionHandler;
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;
use App\Exceptions\EmailNotProvidedException;

class Handler extends ExceptionHandler
{
    /**
     * A list of the exception types that should not be reported.
     *
     * @var array
     */
    protected $dontReport = [
        \Illuminate\Auth\AuthenticationException::class,
        \Illuminate\Auth\Access\AuthorizationException::class,
        \Symfony\Component\HttpKernel\Exception\HttpException::class,
        \Illuminate\Database\Eloquent\ModelNotFoundException::class,
        \Illuminate\Session\TokenMismatchException::class,
        \Illuminate\Validation\ValidationException::class,
    ];

    /**
     * Report or log an exception.
     *
     * This is a great spot to send exceptions to Sentry, Bugsnag, etc.
     *
     * @param  \Exception  $exception
     * @return void
     */
    public function report(Exception $exception)
    {
        parent::report($exception);
    }

    /**
     * Render an exception into an HTTP response.
     *

```

```
* @param \Illuminate\Http\Request $request
* @param \Exception $exception
* @return \Illuminate\Http\Response
*/
public function render($request, Exception $exception)
{
    switch($exception){

        case $exception instanceof EmailNotProvidedException :

            if ($request->ajax()) {

                return response()->json(['error' => 'Email Not Found'], 500);

            }

            return response()
                ->view('errors.email-not-provided-exception',
                    compact('exception'), 500);

            break;

        default:

            return parent::render($request, $exception);

    }
}

/**
 * Convert an authentication exception into an unauthenticated response.
 *
 * @param \Illuminate\Http\Request $request
 * @param \Illuminate\Auth\AuthenticationException $exception
 * @return \Illuminate\Http\Response
*/
protected function unauthenticated
    ($request, AuthenticationException $exception)
{
```

```
if ($request->expectsJson()) {  
  
    return response()->json(['error' => 'Unauthenticated.'], 401);  
  
}  
  
return redirect()->guest('login');  
  
}  
}
```

So we just changed the render method by adding a switch statement. I set this up for extensibility. By using switch statements, we can easily add custom exceptions in the future, which we will do.

So then in the render method, we just check to see if the exception is an instanceof, in the first case, EmailNotProvidedException:

```
switch($exception){  
  
    case $exception instanceof EmailNotProvidedException :  
  
        if ($request->ajax()) {  
            return response()->json(['error' => 'Email Not Found'], 500);  
        }  
  
        return response()  
        ->view('errors.email-not-provided-exception',  
        compact('exception'), 500);  
  
        break;  
  
    default:  
  
        return parent::render($request, $exception);  
    }  
  
}
```

Then we check to see if it is an ajax request, and if so, return some feedback. That isn't really necessary in this case, but I'm writing it this way because it's a good way for providing feedback when there is the possibility of an ajax call.

You can see that if the exception is an instance of EmailNotProvidedException, then we return:

```
return response()
->view('errors.email-not-provided-exception',
compact('exception'), 500);
```

That would all be one line, had to avoid the wordwrap problem. We send along to the view the exception object via compact, so we can have access to getMessage(). We will see this in action in a moment.

Of course, first we need the view in our errors folder, so let's make email-not-found-exception.blade.php now:

Gist:

[email-not-provided-exception.blade.php](#)

From book:

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

@section('content')

<div class="alert alert-danger alert-dismissible alert-important" role="alert">

<button type="button"
        class="close"
        data-dismiss="alert"
        aria-label="Close">

    <span aria-hidden="true">&times;</span>

</button>

<strong>Oh Snap!</strong>
```

```
Your social provider,  
{{ $exception->getMessage() }},  
  
did not provide an email. Please check the settings on your social  
Provider's account and try again.  
  
</div>  
  
@endsection
```

You can see we are also echoing \$exception->getMessage().

Now let's change the index method in our TestController to the following:

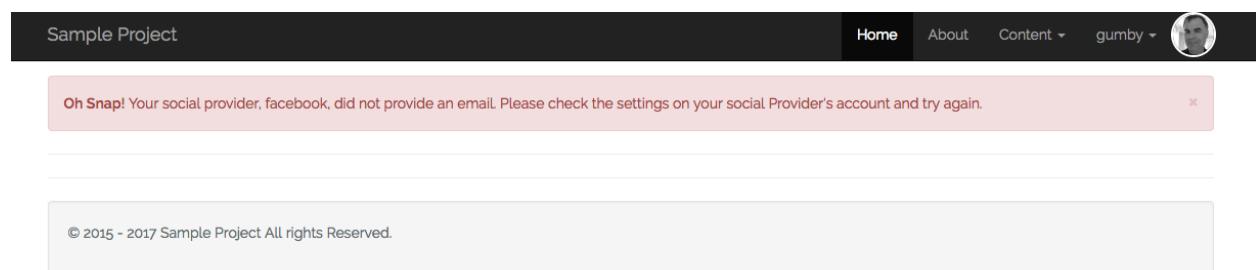
```
public function index()  
{  
  
    throw new EmailNotProvidedException('facebook');  
  
}
```

We'll just throw the exception.

Once again, let's go to:

<http://sample-project.com/test>

And you should get a nicely formatted exception:



The screenshot shows a web application interface. At the top, there is a dark header bar with the text "Sample Project" on the left and navigation links "Home", "About", "Content", and "gumby" on the right, along with a user profile icon. Below the header is a light-colored content area. A red rectangular callout box is positioned in the center of the content area, containing the text "Oh Snap! Your social provider, facebook, did not provide an email. Please check the settings on your social Provider's account and try again." To the right of the callout box is a small "X" button. At the bottom of the page, there is a footer bar with the copyright notice "© 2015 - 2017 Sample Project All rights Reserved."

Summary

This was a big chapter. We built a prototype based on the RESTful pattern that gives us a starting point for other models, including blog posts, faqs, users, profiles and more. We won't always use everything we created here, but it really helps to have it as a point of reference.

In some cases, we will be able to copy all the files in the widget view folder into a new folder and use it as quasi-code generation, a starting point where we simply substitute our new model and input fields, so we save a lot of coding time. We will see that in action later.

We also covered basic error handling, which will come in handy later one when we build our Socialite implementation.

Chapter 7: Access Control

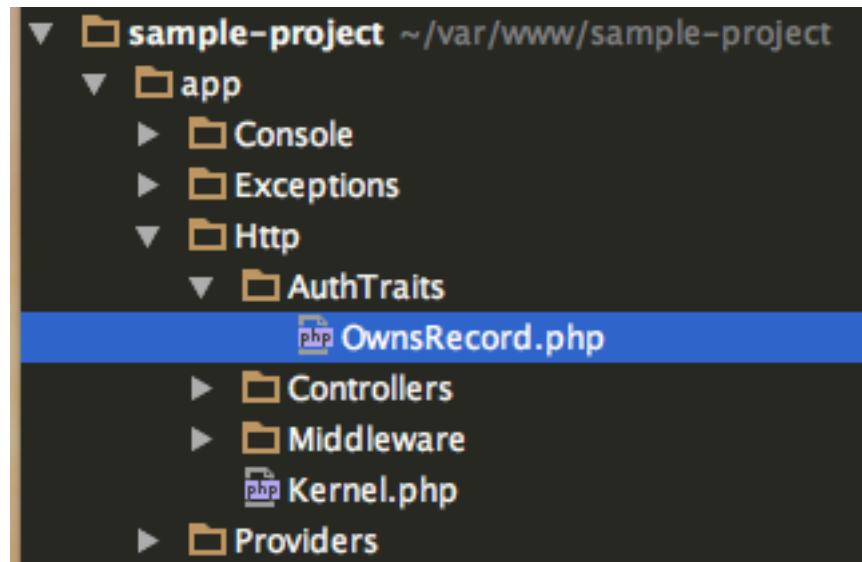
While we can register and login users, we have no other way to control the users. For example, should anyone be able to edit a widget? Or should that be limited to only those who created the widget?

And of course we want to have an admin area to the site, where the admin users can look at stats that are important to the site. So we need to build in that access control as well.

OwnsRecord Trait

We could add an owns method to the user model that will check to see if the user owns the record we are checking, however I prefer to extract this out to a trait. Doing so keeps the model code cleaner and less cluttered.

So let's start by creating a folder in Http named AuthTraits. When we create our OwnsRecord.php file, it will look like this:



Ok, so we need to create OwnsRecord.php within the AuthTraits folder. Create that file now with the following contents.

Gist:

[OwnsRecord.php](#)

From book:

```
<?php  
namespace App\Http\AuthTraits;  
  
use Illuminate\Support\Facades\Auth;  
  
trait OwnsRecord  
{  
  
    public function userNotOwnerOf($modelRecord)  
    {  
  
        return $modelRecord->user_id != Auth::id();  
    }  
  
    public function currentUserOwns($modelRecord)  
    {  
  
        return $modelRecord->user_id === Auth::id();  
    }  
  
    public function adminOrCurrentUserOwns($modelRecord)  
    {  
        if (Auth::user()->is_admin == 1){  
  
            return true;  
        }  
  
        return $modelRecord->user_id === Auth::id();  
    }  
}
```

Ok, you can see that we have our namespace set and that we are using the Auth Facade:

```
namespace App\Http\AuthTraits;  
  
use Illuminate\Support\Facades\Auth;
```

We're going to use Auth::id(), which returns the id of the currently logged in user to test against the user_id attribute on the model we're handing in.

So for example, on the Widget model, we have a user_id, which stores the user who created the record. All we have to do is make sure the id's match and we're good, so you can see this in the currentUserOwns method:

```
public function currentUserOwns($modelRecord)  
{  
  
    return $modelRecord->user_id === Auth::id();  
  
}
```

So we're checking user_id against Auth::id() to see if they are the same user.

Just for fun, I switched the syntax around to run the same test on the userNotOwnerOf method:

```
public function userNotOwnerOf($modelRecord)  
{  
  
    return $modelRecord->user_id != Auth::id();  
  
}
```

So now I could do something like:

```
if ($this->userNotOwnerOf($widget)){  
    // throw new SomeException;  
}
```

Or the same thing with using the other method:

```
if ( ! $this->currentUserOwns($widget)){  
    // throw new SomeException;  
}
```

So I gave us a couple of options, which we can decide on a case by case basis, which one we prefer. That might be overkill, but I like it.

Finally, we have a method that checks to see if the user is either the owner of the record or admin:

```
public function adminOrCurrentUserOwns($modelRecord)  
{  
    if (Auth::user()->is_admin == 1){  
        return true;  
    }  
  
    return $modelRecord->user_id === Auth::id();  
}
```

This will come in really handy if you want to allow admin as well as the owner of a record to be able to edit it.

Now in order for us to be able to see if `is_admin` is equal to one, in other words, true, we have to add that column to the user table.

We will be making some additions to our user table shortly, but before we do that, let's test out our trait. Let's add the trait to our WidgetController:

```
class WidgetController extends Controller
{
    use OwnsRecord;
```

And as a reminder, don't forget to pull it in via use statement in the use statements section:

```
use App\Http\Auth\Traits\OwnsRecord;
```

So let's pop one of trait methods into our update method like so.

Gist:

[Widget Update](#)

From book:

```
public function update(Request $request, $id)
{
    $this->validate($request, [
        'name' => 'required|string|max:30|unique:widgets,name,' . $id
    ]);

    $widget = Widget::findOrFail($id);

    if ($this->userNotOwnerOf($widget)){
        dd('you are not the owner');

    }

    $slug = str_slug($request->name, "-");

    $widget->update([
        'name' => $request->name,
        'slug' => $slug,
        'user_id' => Auth::id()]);
    alert()->success('Congrats!', 'You updated a widget');

    return Redirect::route('widget.show', [
        'widget' => $widget,
        'slug' => $slug
    ]);
}
```

You can see what we added:

```
if ($this->userNotOwnerOf($widget)) {

    dd('you are not the owner');

}
```

So let's test this. If you seeded your DB with records from the updated factory method, just pick a record with an id other than logged in user.

If not, go to your PhpMyadmin and modify the user_id column on a user record to something other than the logged in user. You can do this by clicking on the edit button on the record row:

	id	user_id	widget_name	slug	created_at	updated_at
<input type="checkbox"/>	1	1	everyone wins again	everyone-wins-again	2015-10-27 03:00:51	2015-10-28 03:06:09

In my case, I'm setting it to 3, which is a user that doesn't exist. Now when you try to update that widget, you should get:

"you are not the owner"

This is also another demonstration of how to use the dd() method to validate your way forward to see if you are getting the expected results.

Obviously, in the real application, we can't die and dump here, so what we will do is create a custom error message via an exception.

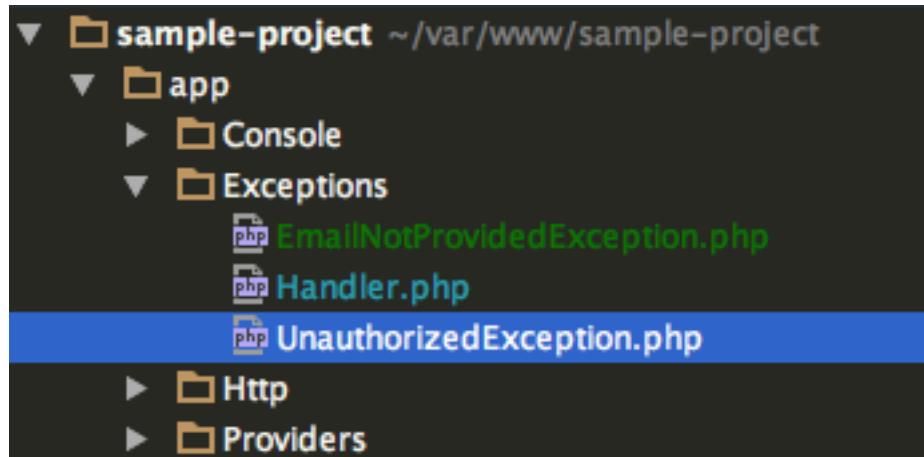
I like to write exceptions with intuitive names. I create different exceptions for different scenarios, so the downside is that I write more exceptions than I would have to if I just wrote a generic exception and passed along a custom message.

As I mentioned in the last chapter, however, it's helpful to have custom exceptions thrown to make troubleshooting easier. There are many 3rd party applications like ErrorStream.com that you can plug into to manage your exception reporting.

We built a custom exception in the last chapter. In workflow, I've made creating an exception simple, using a template approach in 4 simple steps:

1. create the exception
2. modify the Handler file to handle the exception
3. make the view for the exception with the custom message
4. Use the exception in the controller with use statement

Ok, let's start by creating a file in our Exceptions folder called UnauthorizedException, it will look like this when we are done:



Ok, so let's create the UnauthorizedException.php file that goes in that folder with the following:

Gist:

[UnauthorizedException.php](#)

From book:

```
<?php

namespace App\Exceptions;

class UnauthorizedException extends \Exception
{
```

You can see all we are really doing is extending Exception and giving it a friendly name. Next we modify the Handler.php file like so.

Gist:

[Handler.php](#)

From book:

```
<?php
```

```
namespace App\Exceptions;

use Exception;
use Illuminate\Auth\AuthenticationException;
use Illuminate\Foundation\Exceptions\Handler as ExceptionHandler;
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;
use App\Exceptions\EmailNotProvidedException;
use App\Exceptions\UnauthorizedException;

class Handler extends ExceptionHandler
{
    /**
     * A list of the exception types that should not be reported.
     *
     * @var array
     */
    protected $dontReport = [
        \Illuminate\Auth\AuthenticationException::class,
        \Illuminate\Auth\Access\AuthorizationException::class,
        \Symfony\Component\HttpKernel\Exception\HttpException::class,
        \Illuminate\Database\Eloquent\ModelNotFoundException::class,
        \Illuminate\Session\TokenMismatchException::class,
        \Illuminate\Validation\ValidationException::class,
    ];

    /**
     * Report or log an exception.
     *
     * This is a great spot to send exceptions to Sentry, Bugsnag, etc.
     *
     * @param  \Exception  $exception
     * @return void
    }
```



```
        break;

    default:

        return parent::render($request, $exception);

    }

}

/**
 * Convert an authentication exception into an unauthenticated response.
 *
 * @param \Illuminate\Http\Request $request
 * @param \Illuminate\Auth\AuthenticationException $exception
 * @return \Illuminate\Http\Response
 */

protected function unauthenticated
    ($request, AuthenticationException $exception)
{
    if ($request->expectsJson()) {

        return response()->json(['error' => 'Unauthenticated.'], 401);

    }

    return redirect()->guest('login');

}

}
```

So the first thing we did was add the use statement in the use statements section:

```
use App\Exceptions\UnauthorizedException;
```

This is one of those situations where if you don't include that, it doesn't work and you have no idea why. So be extra-careful about use statements when working with your Handler.php file.

From there, all we did is add to our switch statement inside our render method:

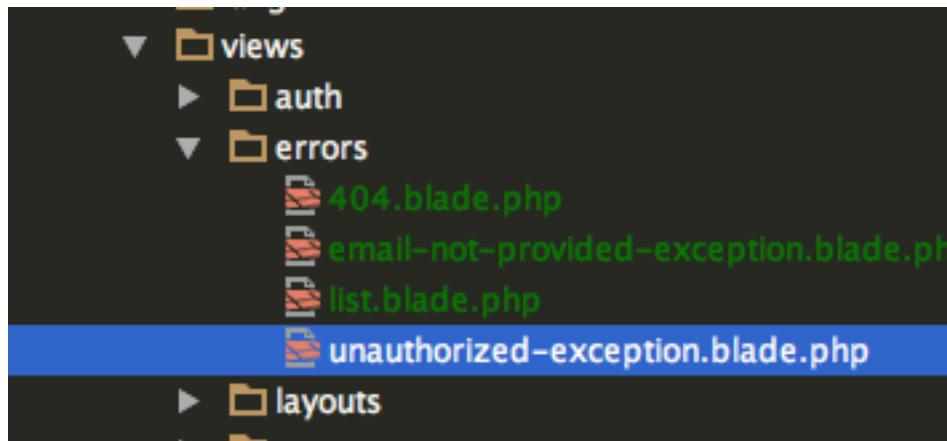
```
case $exception instanceof UnauthorizedException:

    if ($request->ajax()) {
        return response()->json(['error' => 'Unauthorized'], 500);
    }

    return response()
        ->view('errors.unauthorized-exception',
            compact('exception'), 500);

break;
```

Obviously, we do not have an unauthorized.blade.php view in our errors folder, so we need to make that now.



Create unauthorized.blade.php with the following.

Gist:

[unauthorized.blade.php](#)

From book:

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

@section('content')

<div class="alert alert-danger alert-dismissible alert-important" role="alert">

<button type="button"
        class="close"
        data-dismiss="alert"
        aria-label="Close">

    <span aria-hidden="true">&times;</span></button>

<strong>Oh Snap!</strong>

{{ $exception->getMessage() }}

    You are not authorized to do this.

</div>

@endsection
```

I'm adding the call to getMessage so you can optionally pass along a message.

Pretty simple stuff, now let's use it in our controller. Add the use statement to the use statements at the top of the WidgetController:

```
use App\Exceptions\UnauthorizedException;
```

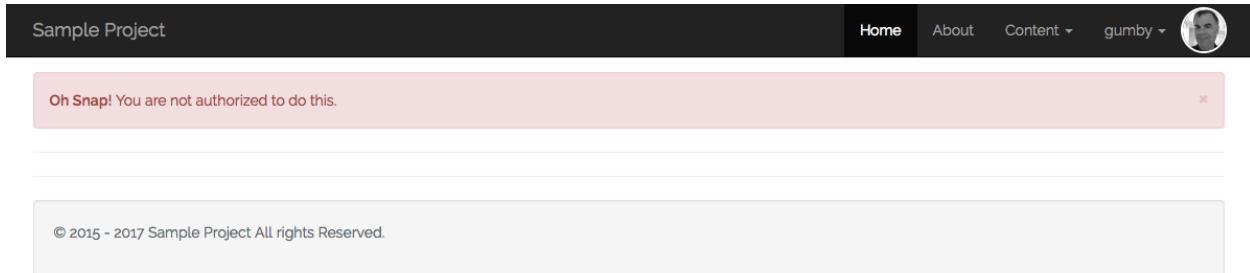
Then in our Update method, remove:

```
dd('you are not the owner');
```

And replace it with:

```
throw new UnauthorizedException;
```

Now if you try updating the widget again, you should see:



I really like how this reads:

```
if ($this->userNotOwnerOf($widget)){  
    throw new UnauthorizedException;  
}
```

I know if I come back to this code in a year, I will know exactly what it is. Super-simple stuff.

So you might be saying to yourself, hmm. Wouldn't it be better not to show users the edit link if they are not authorized? Well, actually, you're right, we should do both.

But to do this, we need to use our OwnsRecord trait on the User model. We also probably want to let our admin users be able to change widgets that they don't own. That means we need to modify our User model to account for admin, and now is a good time to do that.

Modify User Table Migration

Let's go to the command line and run the following, but, obviously, do it as one line not two:

```
php artisan make:migration  
add_is_admin_and_status_id_columns_to_users_table
```

So we should now have that file in our database/migrations folder, with empty stubs for the up and down methods. Let's modify it to the following.

Gist:

[migration](#)

From book:

```
<?php  
  
use Illuminate\Support\Facades\Schema;  
use Illuminate\Database\Schema\Blueprint;  
use Illuminate\Database\Migrations\Migration;  
  
class AddIsAdminAndStatusIdColumnsToUsersTable extends Migration  
{  
    /**  
     * Run the migrations.  
     *  
     * @return void  
     */  
  
    public function up()  
    {  
        Schema::table('users', function(Blueprint $table)  
        {  
  
            $table->boolean('is_admin')->default(false)->after('email');
```

```
$table->integer('status_id')
    ->default(10)
    ->unsigned()
    ->after('is_admin');

});

/** 
 * Reverse the migrations.
 *
 * @return void
 */
public function down()
{
    Schema::table('users', function ($table) {
        $table->dropColumn('is_admin');
        $table->dropColumn('status_id');

    });
}

}
```

Ok, so you can see we are adding 2 columns. The first one is `is_admin`, a simple boolean that defaults to false and is placed after the `email` column.

```
$table->boolean('is_admin')->default(false)->after('email');
```

I really like being able to specify column order, that's a nice feature of Laravel's migrations.

The [migration docs](#) are very comprehensive, so if you need to learn how to modify a column for example, don't be shy about checking the docs first.

Next we have the status_id column. I'm naming the column as if it were going to use a foreign key, but we won't be doing that. Instead we will simply give it a value, and in this case, the default is value is 10.

So why name it this way? If in the future we decide to build out statuses into a model, let's say we have a bunch of different statuses we want to hold in its own table, the data structure on the users table already supports it. Just thinking ahead a little in case we need it.

```
$table->integer('status_id')->default(10)->unsigned()->after('id');
```

If later we want to build a status table, the users data structure already supports it.

For this project however, we will be keeping it simple. There will only be 2 user types and only 2 statuses, so we will not be building separate tables for those.

Obviously the down method is set up to drop the columns:

```
Schema::table('users', function ($table) {  
  
    $table->dropColumn('is_admin');  
    $table->dropColumn('status_id');  
  
});
```

Ok, let's run the migration from the command line:

```
php artisan migrate
```

And now your table should look like this:

So you can see how incredibly easy this workflow is. We modified the users table with no difficulty at all.

User \$fillable

Let's add the new attributes to the \$fillable property on the User model.

Gist:

[User \\$fillable](#)

From book:

```
protected $fillable = ['name',
                      'email',
                      'is_admin',
                      'status_id',
                      'password'];
```

This will allow us to mass-assign those values.

Ok, before we move on to making some deeper changes for admin, let's do one last test for our OwnsRecord trait.

We'll start by modifying our is_admin value to 1 on our user via PhpMyAdmin:

+ Options							
	← →	id	name	email	is_admin	status_id	password
<input type="checkbox"/>	Edit	Copy	Delete	1	gumby	sample@smaple.com	1
<input checked="" type="checkbox"/>	Check All	With selected:		Change	Delete	Export	\$2y\$10\$R0bf8d6o70jgYDJsVPMhK.oHN1Ag

Next, let's go back to the update method on our Widget controller and change the check for owner of record to the following:

```
if ( ! $this->adminOrCurrentUserOwns($widget)){
    throw new UnauthorizedException;
}
```

Once that change is in place, try updating a widget.

This time, even though you are not the owner of the record, because you are admin, you are able to update the record. This is incredibly easy to work with.

Something to note. The way we have this set up, the user_id gets updated to the Auth::user(), so it changes to your id when you update. If you don't want it to do that, you need to change the update method to reflect that. For example, you could do it by eliminating that field from the update method.

So now, we want to exert the same control over the view. If the user does not own the record or is not admin, they will not get a link to the edit form.

The first step in making this work is to pull in our OwnsRecord trait into our User model. So at the top, in the use statements section, add:

```
use App\Http\Auth\Traits\OwnsRecord;
```

Then use OwnsRecord in the class:

```
namespace App;

use App\Http\Auth\Traits\OwnsRecord;
use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use Notifiable, OwnsRecord;
```

Now we can call our methods via Auth::user(), such as Auth::user->adminOrCurrentUserOwns().

Let's go to our show view, and modify it as follows. I'll give you the gist of the entire show.blade.php file for reference, but I will only show the changed part in the book version.

Gist:

[show.blade.php](#)

From book:

```
<!-- Table -->



| ID | Name | Date Created | Edit | Delete |
|----|------|--------------|------|--------|
|----|------|--------------|------|--------|


```

```
</tr>

</thead>

<tbody>

<tr>
  <td>

    {{ $widget->id }}

  </td>
  <td>

    <a href="/widget/{{ $widget->id }}/edit">
      {{ $widget->name }}
    </a>
  </td>
  <td>

    {{ $widget->created_at }}

  </td>

  @if(Auth::user()->adminOrCurrentUserOwns($widget))
  <td>

    <a href="/widget/{{ $widget->id }}/edit">
      <button type="button"
        class="btn btn-default">
        Edit
      </button>
    </a>
  </td>

  @endif
```

```
<td>
  <div class="form-group">

    <form class="form"
      role="form"
      method="POST"
      action="{{ url('/widget/'. $widget->id) }}>

      <input type="hidden" name="_method" value="delete">

      {{ csrf_field() }}

      <input class="btn btn-danger"
        Onclick="return ConfirmDelete();"
        type="submit"
        value="Delete">

    </form>
  </div>

</td>
</tr>
</tbody>
</table>
```

Please use the gist for code formatting.

So now as long as your user is not admin and the widget has a different user_id value than the user id, you will not see the link. Play around with your is_admin value in your DB and you can see how it works.

You can see that we chained our methods:

@if(Auth::user()->adminOrCurrentUserOwns(\$widget))

And that's pretty easy to follow. If admin or owner, show the link, if not, don't show the link.

Of course, you also have to protect the form because someone could just type in the get request for the form in the url and it would display. We don't want that.

So in WidgetController.php, change the edit method to the following:

Gist:

[edit method](#)

From book:

```
public function edit($id)
{
    $widget = Widget::findOrFail($id);

    if ( ! $this->adminOrCurrentUserOwns($widget)){
        throw new UnauthorizedException;
    }

    return view('widget.edit', compact('widget'));
}
```

And that will protect the edit form from being accessed directly by anyone who shouldn't be seeing it.

So you can see that with a simple trait, and modification to our User model, we were able to exert a good amount of control over our users.

It would make a lot of sense to control the delete button in the same way as edit. I will leave it up to you if you want to implement that.

You may be asking yourself, what about situations where we only want to limit access to admin users? Well, we could test for `is_admin == 1` all over the place, but that wouldn't be efficient. Fortunately, Laravel has a much better solution.

Admin Middleware

Laravel's middleware is a very cool part of the framework. I was completely unfamiliar with it when I started with Laravel, so I was a little put off by it initially, but once I got used to it, I really liked it.

We briefly went over middleware in chapter 6, when we looked at how we enforce a user's authentication, making sure they are logged in for various parts of the site. That middleware comes straight out of the box. All we had to do was pop it into the controller's constructor and we are good to go.

I should also note that there are other ways to use middleware. As an alternative to using it through the constructor, you can place it directly on the routes like so:

```
Route::get('test', 'TestController@index')->middleware('auth');
```

If you make that update to your test route in routes/web.php, you will require login to get to our test page.

You could also make the middleware value an array of values:

```
Route::get('test', 'TestController@index')->middleware(['auth', 'throttle']);
```

We don't have an admin middleware yet, but we will shortly. Laravel has [excellent documentation](#) on it's middleware, so check that out when you get a chance.

Note: In the docs, in a lot of their examples, they use a closure, but I prefer the simpler syntax that I've given above. The fluent syntax is new in Laravel 5.4.

I find that when I'm working with middleware, I generally tend to use it in the constructor. That's not a hard rule of course, but it has been my experience so far.

So the next step in learning about middleware is to actually make one. We are going to create an admin middleware, that will enforce whether or not a user is admin.

AllowIfAdmin

The good news is that artisan has a very handy command to create a middleware stub. From the command line, run:

```
php artisan make:middleware AllowIfAdmin
```

And that should add AllowIfAdmin.php to the app/Http/Middleware folder. Before we look at it, let's add the call from app/Http/Kernel.php. We'll add it as the last line in the protected \$routeMiddleware array:

```
protected $routeMiddleware = [  
  
    'auth' => \Illuminate\Auth\Middleware\Authenticate::class,  
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,  
    'bindings' => \Illuminate\Routing\Middleware\SubstituteBindings::class,  
    'can' => \Illuminate\Auth\Middleware\Authorize::class,  
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,  
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,  
    'admin' => \App\Http\Middleware\AllowIfAdmin::class,  
  
];
```

I gave you the full array for reference, but you only need to add the last line. When we call our middleware, we will use ‘admin’ to call it and it will refer to the AllowIfAdmin class. Simple enough.

Ok, let’s see what artisan made for us. Let’s open AllowIfAdmin.php:

```
<?php  
  
namespace App\Http\Middleware;  
  
use Closure;  
  
class AllowIfAdmin  
{  
    /**  
     * Handle an incoming request.  
     *  
     * @param \Illuminate\Http\Request $request  
     * @param \Closure $next  
     * @return mixed  
     */  
  
    public function handle($request, Closure $next)  
    {  
  
        return $next($request);  
    }  
}
```

Ok, not a lot to this yet. We have a stub for the handle method. This is a good place to mention that there is before middleware and after middleware.

In before middleware, you are going to do your work before the request processes:

```
public function handle($request, Closure $next)
{
    //your code goes here

    return $next($request);
}
```

So in our case, our code will be a test for admin. Before we get to that, let's look at after middleware.

In after middleware, you do it after the request:

```
public function handle($request, Closure $next)
{
    $response = $next($request);

    // your code goes here

    return $response;
}
```

Just as a heads up, there's a great series on Laracasts about implementing pjax to render pages, which is outside the scope of this book, but it uses after middleware in those videos. Check that out if you want to see a working example. We are not using after middleware in this book, I just wanted to point it out to you, so you are aware.

Ok, we're ready to modify AllowIfAdmin.php:

Gist:

[AllowIfAdmin.php](#)

from book:

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Support\Facades\Auth;
use App\Exceptions\UnauthorizedException;

class AllowIfAdmin
{
    /**
     * Handle an incoming request.
     *
     * @param \Illuminate\Http\Request $request
     * @param \Closure $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        if (Auth::check() && Auth::user()->status_id == 10) {

            if (Auth::user()->is_admin == 1) {

                return $next($request);

            }

        }

        throw new UnauthorizedException;
    }
}
```

Ok, let's look at what we have here. You can see we added a couple of use statements:

```
use Illuminate\Support\Facades\Auth;
use App\Exceptions\UnauthorizedException;
```

We've seen the Auth facade before, it gives us access to the currently authenticated user, including the properties and methods on the user model.

We also have the UnauthorizedException that we created.

```
public function handle($request, Closure $next)
{
    if (Auth::check() && Auth::user()->status_id == 10) {

        if (Auth::user()->is_admin == 1) {

            return $next($request);

        }

    }

    throw new UnauthorizedException;
}
```

Ok, so we have a nested if statement. The first if statement determines if the user is logged in and has a status_id of 10. If you remember we set 10 as the default, and in our template, that represents an active user.

The 2nd if simply checks to see if the user's is_admin record is set to 1. If it is, they are admin and we can proceed. If not, we throw an UnauthorizedException.

So what may have looked complicated at first glance, is actually incredibly simple.

Now to check and see if this working, let's add the middleware to our widget controller, like so:

```
public function __construct()
{
    $this->middleware(['auth', 'admin'], ['except' => ['index', 'show']]);
}
```

You can see that we hand in the first parameter as an array, so we can list both middlewares we want to apply there because we want to apply them to the same methods. If you want to apply the middleware to different methods, you will need two separate lines.

So this will require auth and admin status on all methods, except index and show,. Go ahead and test it with a logged in user who is not admin.

Try this url:

```
http://my-sample-project.com/widget/1/edit
```

If you have logged in with a user that is not admin, you should not be able to get to that view.

Let's change the constructor to the following:

```
public function __construct()
{
    $this->middleware('auth', ['except' => 'index']);
    $this->middleware('admin', ['except' => ['index', 'show']]);
}
```

Now we are using two middleware statements to fine tune what we want. Note that we are requiring login on the show method because later we will test for admin in the show method and we can't do that if the user is not logged in.

Also note: If you were previously logged in with a different admin value, logout and then back in for testing.

So this is great, we now have a middleware that enforces admin status.

Although the middleware concept may have been a little foreign, you can see that the actual implementation is extremely simple. We can easily think of which middleware applies to a controller and then simply put it in there.

When I want to know who has access to what, I just look at the constructor and that gives me the big picture. We use other methods, like the ones from the OwnsRecord trait, for a more granular approach to controlling access.

isAdmin Method

We can see from our middleware implementation that we are using the following:

```
if ($this->auth->user()->is_admin == 1)
```

That's fairly clear, but it could be better. Why don't we make a simple method on the User model named isAdmin that let's us make it simpler. Add the following use statement to your User model:

```
use Illuminate\Support\Facades\Auth;
```

Then add the following method:

```
public function isAdmin()
{
    return Auth::user()->is_admin == 1;
}
```

It's a one line method, so no gist.

So now we can rewrite that line in our admin middleware as follows:

```
if (Auth::user()->isAdmin())
```

That's little easier to digest, and we can use isAdmin in our views as well.

In fact let's do one more for the status_id check, since this syntax is so nice. Let's create the following method in our User.php model:

```
public function isActiveStatus()
{
    return Auth::user()->status_id == 10;
}
```

So now we can take the following the line in our AllowIfAdmin middleware:

```
if ($this->auth->check() && $this->auth->user()->status_id == 10)
```

And change it to:

```
if (Auth::check() && Auth::user()->isActiveStatus())
```

So you can see it's an improvement in readability. Plus obviously if you decide to change what the active status value is, you are only changing it in one place.

For reference, so you can check it if you have errors, I'll give you a gist of the new AllowIfAdmin.php file:

[AllowIfAdmin.php](#)

One thing to think about. Because I included the Auth::check() and Auth::user()->isActiveStatus(), I can just use one line of middleware to check for that and admin.

If you feel it's too much in one file, you can extract out isActiveStatus into it's own middleware.

Admin Index

So now what we need is a place to land admin users when they login. Logically, we would have special admin area for management of the site, and we want admin users to go straight to it.

The other thing we want to do is when a user attempts to login, check to see if their status_id is set to 10. If they don't have that, then they are not an active user.

If they are not active, we will throw an exception, we'll create a NoActiveAccountException.

Since we'll be using the new exception, why don't we start by creating that.

NoActiveAccountException

If you remember from earlier in the chapter, we got this down to 4 steps:

1. Create the exception
2. Modify the Handler file to handle the exception
3. Make the view for the exception with the custom message
4. Use the exception in the controller with use statement

Since we've been through this already, I'm going to move quickly. Let's create NoActiveAccountException.php in the app/Exceptions folder with the following:

Gist:

[NoActiveAccountException.php](#)

From book:

```
<?php

namespace App\Exceptions;

class NoActiveAccountException extends \Exception
{
```

Then we will change the app/Exceptions/Handler.php file to the following:

Gist:

[Handler.php](#)

From book:

```
<?php

namespace App\Exceptions;

use Exception;
use Illuminate\Auth\AuthenticationException;
use Illuminate\Foundation\Exceptions\Handler as ExceptionHandler;
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;
use App\Exceptions\EmailNotProvidedException;
use App\Exceptions\UnauthorizedException;
use App\Exceptions\NoActiveAccountException;

class Handler extends ExceptionHandler
{
    /**
     * A list of the exception types that should not be reported.
     *
     * @var array
     */

    protected $dontReport = [
        \Illuminate\Auth\AuthenticationException::class,
        \Illuminate\Auth\Access\AuthorizationException::class,
        \Symfony\Component\HttpKernel\Exception\HttpException::class,
        \Illuminate\Database\Eloquent\ModelNotFoundException::class,
        \Illuminate\Session\TokenMismatchException::class,
        \Illuminate\Validation\ValidationException::class,
    ];

    /**
     * Report or log an exception.
     *
     * This is a great spot to send exceptions to Sentry, Bugsnag, etc.
    
```

```
*  
* @param \Exception $exception  
* @return void  
*/  
  
public function report(Exception $exception)  
{  
    parent::report($exception);  
}  
  
/**  
 * Render an exception into an HTTP response.  
 *  
 * @param \Illuminate\Http\Request $request  
 * @param \Exception $exception  
 * @return \Illuminate\Http\Response  
*/  
  
public function render($request, Exception $exception)  
{  
  
    switch($exception){  
  
        case $exception instanceof EmailNotProvidedException :  
  
            if ($request->ajax()) {  
  
                return response()->json(['error' => 'Email Not Found'], 500);  
            }  
  
            return response()  
                ->view('errors.email-not-provided-exception',  
                    compact('exception'), 500);  
  
        break;  
  
        case $exception instanceof NoActiveAccountException:  
  
            if ($request->ajax()) {  
  
                return response()->json(['error' => 'No Active Account'], 500);  
            }  
    }  
}
```

```
    return response()
        ->view('errors.no-active-account-exception',
            compact('exception'), 500);

    break;

case $exception instanceof UnauthorizedException:

    if ($request->ajax()) {

        return response()->json(['error' => 'Unauthorized'], 500);

    }

    return response()
        ->view('errors.unauthorized-exception',
            compact('exception'), 500);

    break;

default:

    return parent::render($request, $exception);

}

}

/** 
 * Convert an authentication exception into an unauthenticated response.
 *
 * @param \Illuminate\Http\Request $request
 * @param \Illuminate\Auth\AuthenticationException $exception
 * @return \Illuminate\Http\Response
 */

protected function unauthenticated
    ($request, AuthenticationException $exception)
{
    if ($request->expectsJson()) {

        return response()->json(['error' => 'Unauthenticated.'], 401);

    }
}
```

```
    }

    return redirect()->guest('login');

}

}
```

You're just adding the case statements. Note that we have our use statement at the top. Also note that I put the case statements in alphabetical order according to exception type, which makes it easier to find something in the file.

Next, let's create the view that we are calling from the renderException method, which is no-active-account.blade.php. Note the dashes in the name to make it more readable.

Gist:

[no-active-account.blade.php](#)

From book:

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

@section('content')

<div class="container">

<div class="alert alert-danger alert-dismissible alert-important"
role="alert">

<button type="button"
class="close"
data-dismiss="alert"
aria-label="Close">

<span aria-hidden="true">&times;</span>

</button>

<strong>Oh Snap! </strong>
```

```
 {{ $exception->getMessage() }}  
  
You do not have an active account.  
  
</div>  
  
</div>  
  
@endsection
```

Remember to use the gist for formatting.

Ok, so now for our LoginController, we can put the use statement in the use statement section:

```
use App\Exceptions\NoActiveAccountException;
```

Modifying the LoginController

Now we're going to make some modifications to the Login controller, and it's worth noting the reasoning behind our approach to it. I'm going to pull a couple of methods up from the trait and overwrite them on the controller itself.

This means the original trait, which resides in the Illuminate namespace deep in the vendor directory, will be intact and untouched. Why does that matter?

Well, as we discussed earlier in the book, what happens when you run composer update is that the vendor files will be replaced with newer files when upgrades to the framework are released and when dependencies are updated. This would overwrite any changes you made there.

One answer is to write your own traits and completely detach from the updates, but this has a downside as I learned from personal experience.

When laravel 5 first came out, I rewrote the auth controller, so that it didn't rely on the traits. Big mistake. Within 2 months, they came out with the ThrottlesLogin trait, which is very useful for fighting off login attacks against your site. You'll want to take advantage of that.

As the framework evolves, they introduce new features and you are going to want to use them. My approach will still break if they change something in the underlying code that causes it to break, but the upside is that it forces you to stay on top of it. I prefer to go in that direction when working with framework code.

Ok, let's get started. In thinking about this logically, before working on the LoginController, it makes sense to create the destination first. We need a landing page for Admin. We also need the route and controller, so let's start with the route:

```
Route::get('/admin', 'AdminController@index')->name('admin');
```

So we are naming the route as admin to make it easier to work with, specifying that it uses the AdminController@index method. Of course we don't have that controller, so why don't we make it now.

From the command line, run:

```
php artisan make:controller AdminController
```

For now, let's just return an index view in the index method like so:

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

Your controller should look like this:

Gist:

[AdminController](#)

From book:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class AdminController extends Controller
{

    public function __construct()
    {

        $this->middleware(['auth', 'admin']);

    }

    public function index()
    {

        return view('admin.index');

    }

}
```

You can see we have everything in place to limit users of this route to logged in admin users. Of course we don't have the index view, so let's make an admin folder in the views directory and place the following index.blade.php within it:

Gist:

[index.blade.php](#)

From book:

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

@section('title')
    <title>The Admin Page</title>

@endsection

@section('content')
    <h1>I Am Admin</h1>
    
@endsection
```

So this is just a placeholder, so we can see we are landing in the right place. Go ahead and check it in the browser by pointing your browser to:

`my-sample-project.com/admin`

Just a reminder, you will need to be logged in with a user that has `is_admin` set to 1 to be able to see the page. Ok, that should all work just fine.

Now we want to set up two things.

1. We want to require `status_id == 10` on the user record for successful login.
2. We want to redirect admin users to the admin index page on login.

To do this, we need to bring up and modify two methods from the traits directly into the `LoginController`, as well as add one new method. And of course add some use statements.

LoginController

I will give you the complete file and then we will discuss each change.

Gist:

[LoginController.php](#)

From book:

```
<?php

namespace App\Http\Controllers\Auth;

use App\Http\Controllers\Controller;
use Illuminate\Foundation\Auth\AuthenticatesUsers;
use App\Exceptions\NoActiveAccountException;
use Illuminate\Support\Facades\Auth;
use Illuminate\Http\Request;

class LoginController extends Controller
{
    /*
    |--------------------------------------------------------------------------
    | Login Controller
    |--------------------------------------------------------------------------
    |
    | This controller handles authenticating users for the application and
    | redirecting them to your home screen. The controller uses a trait
    | to conveniently provide its functionality to your applications.
    |
    */
    use AuthenticatesUsers;

    /**
     * Where to redirect users after login.
     *
     * @var string
     */

    protected $redirectTo = '/';

    /**
     * Create a new controller instance.
     *
     * @return void
     */

    public function __construct()
    {
        $this->middleware('guest', ['except' => 'logout']);
    }
}
```

```
}

private function checkStatusLevel()
{
    if ( ! Auth::user()->isActiveStatus() ) {

        Auth::logout();

        throw new NoActiveAccountException;

    }

}

public function redirectPath()
{
    if (Auth::user()->isAdmin()){

        return 'admin';
    }

    return property_exists($this, 'redirectTo') ? $this->redirectTo : '/';
}

public function login(Request $request)
{
    $this->validateLogin($request);

    // If the class is using the ThrottlesLogins trait,
// we can automatically throttle
// the login attempts for this application. We'll key this
// by the username and
// the IP address of the client making these requests
// into this application.

    if ($this->hasTooManyLoginAttempts($request)) {
        $this->fireLockoutEvent($request);

        return $this->sendLockoutResponse($request);
    }
}
```

```
if ($this->attemptLogin($request)) {  
  
    $this->checkStatusLevel();  
  
    return $this->sendLoginResponse($request);  
}  
  
// If the login attempt was unsuccessful we will  
// increment the number of attempts  
// to login and redirect the user back to the login form.  
// Of course, when this  
// user surpasses their maximum number of attempts  
// they will get locked out.  
  
$this->incrementLoginAttempts($request);  
  
return $this->sendFailedLoginResponse($request);  
}  
}
```

Ok, so we're only going to cover what is new. To start with, we added some use statements:

```
use App\Exceptions\NoActiveAccountException;  
use Illuminate\Support\Facades\Auth;  
use Illuminate\Http\Request;
```

We copied the login method from the AuthenticatesUsers trait and overwrote it on the AuthController to include this line:

```
$this->checkStatusLevel();
```

I kept the change as minimal and unobtrusive as possible by extracting out to a method:

```
private function checkStatusLevel()
{
    if ( ! Auth::user()->isActiveStatus() ) {

        Auth::logout();

        throw new NoActiveAccountException;

    }
}
```

So you can see what we are doing here is checking the status of the authenticated user, and if they are not active, we log them out via `Auth::logout()`, another handy Auth method.

Inactive users also get the `NoActiveAccountException`, which we made earlier.

This login method that we used is current as of Laravel 5.4.0. Laravel tends to release minor versions often and this method could change in a subsequent release. If that is the case, and I have not yet updated the book, copy the login method from the `AuthenticatesUsers` Traits instead of what I'm providing above. That said, I will try to stay on top of it, so you don't have to worry about it.

Finally, we overwrote the `redirectTo` method:

```
public function redirectPath()
{
    if (Auth::user()->isAdmin()){

        return 'admin';
    }

    return property_exists($this, 'redirectTo') ? $this->redirectTo : '/';
}
```

So we just check to see if the user is admin, and if so, return ‘admin’, which is the name of the route we want. If not admin, then go to the redirectTo property if it exists or default ot ‘/’.

These are only minor changes, so they are easy to maintain in terms of anything changing with the framework, and yet, they give us a lot of functionality that we desired. If for example, the next version of laravel has a change to the login method in the trait, which is entirely likely, you will only have to change one method when you update to the latest version.

Anyway, let’s get back to what we have now. You can play around with different user accounts and login to test everything. If you are admin, you will get the admin index page. If not, you will get the pages index page.

You can also play with your status_id. If you set it to anything other than 10, you will get the NoActiveAccountException.

Update Users Table

Now that we enhanced our application’s login method, so that it checks for status_id and whether or not the user is_admin, we need to make a couple of changes for registration.

What we have now is fine, but there are two glaring omissions.

1. We need to ask for terms of service agreement
2. We need to ask if they wish to receive our newsletter.

Not every application will require a newsletter subscription, but it’s good to know how to do this anyway. So let’s dig in.

Registration Form

We’ll start by making a couple of changes to register.blade.php, which is in our auth folder in views.

register.blade.php

So let's go ahead and change register.blade.php to the following. Remember to use the Gist for code style and format:

Gist:

[register.blade.php](#)

From book:

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

@section('content')



- <a href="/">Home</a></li>
    - Register</li>



Register


```

```
<div class="col-md-6">

<input id="name"
       type="text"
       class="form-control"
       name="name"
       value="{{ old('name') }}" autofocus>

@if ($errors->has('name'))

<span class="help-block">

<strong>{{ $errors->first('name') }}</strong>

</span>

@endif

</div>
</div>

<div class="

form-group{{ $errors->has('email') ? ' has-error' : '' }}>

">

<label for="email"
      class="col-md-4 control-label">

    E-Mail Address

</label>

<div class="col-md-6">

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

@if ($errors->has('email'))
```

```
<span class="help-block">

<strong>{{ $errors->first('email') }}</strong>

</span>

@endif

</div>

</div>

<div class="

    form-group{{ $errors->has('password') ? ' has-error' : '' }}>

    " >

<label for="password"
      class="col-md-4 control-label">

        Password

    </label>

<div class="col-md-6">

<input id="password"
      type="password"
      class="form-control"
      name="password">

@if ($errors->has('password'))

<span class="help-block">

<strong>{{ $errors->first('password') }}</strong>

</span>

@endif

</div>
</div>
```

```
<div class="form-group{{ $errors->has('password_confirmation') ? ' has-error' : '' }}>

">

<label for="password-confirm"
       class="col-md-4 control-label">

    Confirm Password

</label>

<div class="col-md-6">

<input id="password-confirm"
       type="password"
       class="form-control"
       name="password_confirmation">

@if ($errors->has('password_confirmation'))

<span class="help-block">

<strong>{{ $errors->first('password_confirmation') }}</strong>

</span>

@endif

</div>
</div>

<div class="form-group">

<label class="col-md-4 control-label">

Subscribe to Newsletter?

</label>

<div class="col-md-6">
```

```
<input type="checkbox" name="is_subscribed">

</div>
</div>

<div class="form-group">

<label class="

    col-md-4 control-label{{ $errors->has('terms') ? ' has-error' : '' }}>

    " >

<a href="/terms-of-service">

    Agree To Terms

</a></label>

<div class="col-md-6">

<input type="checkbox" name="terms" required>

@if ($errors->has('terms'))

<span class="help-block"><strong>

{{ $errors->first('terms') }}</strong></span>

@endif

</div>
</div>

<div class="form-group">
<div class="col-md-6 col-md-offset-4">

<button type="submit"
       class="btn btn-primary">

    Register

</button>
</div>
</div>
```

```
</button>

</div>
</div>

</form>

</div>
</div>
</div>
</div>
</div>

@endsection
```

The two differences to the form are:

```
<div class="form-group">

<label class="col-md-4 control-label">
    Subscribe to Newsletter?
</label>

<div class="col-md-6">
    <input type="checkbox" name="is_subscribed">
</div>
</div>

<div class="form-group">
    <label class="col-md-4 control-label {{ $errors->has('terms') ? ' has-error' : '' }}>
```

```
<a href="/terms-of-service">  
    Agree To Terms  
</a>  
</label>  
  
<div class="col-md-6">  
    <input type="checkbox" name="terms" required>  
    @if ($errors->has('terms'))  
        <span class="help-block">  
            <strong>{{ $errors->first('terms') }}</strong>  
        </span>  
    @endif  
</div>  
</div>
```

You can see we will be passing along is_subscribed and terms from the checkboxes.

But of course that will not work now, we don't have an is_subscribed column on our users table.

Users Migration

So let's modify our users table with a migration. We'll run the following artisan command from the command line:

```
php artisan make:migration add_is_subscribed_to_users_table
```

That will get you the migration file, which you can then modify to the following.

Gist:

[Users Migration](#)

From book:

```
<?php

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

class AddIsSubscribedToUsersTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */

    public function up()
    {
        Schema::table('users', function(Blueprint $table)
        {

            $table->boolean('is_subscribed')
                ->default(false)->after('email');

        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */

    public function down()
    {
        Schema::table('users', function ($table) {
            $table->dropColumn('is_subscribed');
        });
    }
}
```

```

    }
}

```

You can see we are placing the new `is_subscribed` column after the `email` column and defaulting it to false.

Let's run the following from the command line:

```
php artisan migrate
```

Now if you check your PhpMyadmin users table, you should see:

	#	Name	Type	Collation	Attributes	Null	Default	Extra	
<input type="checkbox"/>	1	<code>id</code>	int(10)		UNSIGNED	No	<code>None</code>	AUTO_INCREMENT	Change
<input type="checkbox"/>	2	<code>name</code>	varchar(255)	utf8_unicode_ci		No	<code>None</code>		Change
<input type="checkbox"/>	3	<code>email</code>	varchar(255)	utf8_unicode_ci		No	<code>None</code>		Change
<input type="checkbox"/>	4	<code>is_subscribed</code>	tinyint(1)			No	0		Change
<input type="checkbox"/>	5	<code>is_admin</code>	tinyint(1)			No	0		Change
<input type="checkbox"/>	6	<code>user_type_id</code>	int(10)		UNSIGNED	No	10		Change
<input type="checkbox"/>	7	<code>status_id</code>	int(10)		UNSIGNED	No	10		Change
<input type="checkbox"/>	8	<code>password</code>	varchar(60)	utf8_unicode_ci		No	<code>None</code>		Change
<input type="checkbox"/>	9	<code>remember_token</code>	varchar(100)	utf8_unicode_ci		Yes	<code>NULL</code>		Change
<input type="checkbox"/>	10	<code>created_at</code>	timestamp			No	0000-00-00 00:00:00		Change
<input type="checkbox"/>	11	<code>updated_at</code>	timestamp			No	0000-00-00 00:00:00		Change

Up Check All With selected: Browse Change Drop Primary Unique Index

Update User \$fillable

Let's add the new `is_subscribed` property to the fillable array:

```
protected $fillable = ['name',
                      'email',
                      'is_subscribed',
                      'is_admin',
                      'user_type_id',
                      'status_id',
                      'password'];
```

Now that the new property is mass-assignable, we need to make some changes to the AuthController.

RegisterController

We'll start with the validator method. It takes in an array of data from the form post, which is what we want, but we need to format the is_subscribed and terms values into boolean.

Gist:

[validator method](#)

From book:

```
protected function validator(array $data)
{
    $data['is_subscribed'] = empty($data['is_subscribed']) ? 0 : 1;
    $data['terms'] = empty($data['terms']) ? 0 : 1;

    return Validator::make($data, [
        'name' => 'required|max:255',
        'email' => 'required|email|max:255|unique:users',
        'is_subscribed' => 'boolean',
        'password' => 'required|min:6|confirmed',
        'terms' => 'accepted'
    ]);
}
```

You can see we've used a ternary to set the value of `is_subscribed` and terms to either 0 or 1 depending on the input from the checkbox.

Then we use the validator to compare the values and pass or fail validation. Note the use of the 'accepted' validator. That means that it can only be true, since you must accept terms of service. How cool is that?

Create Method

Ok, let's move on to our create method.

Gist:

[create method](#)

From book:

```
protected function create(array $data)
{
    $data['is_subscribed'] = empty($data['is_subscribed']) ? 0 : 1;

    return User::create([
        'name' => $data['name'],
        'email' => $data['email'],
        'is_subscribed' => $data['is_subscribed'],
        'password' => bcrypt($data['password']),
    ]);
}
```

We do the formatting on the `is_subscribed` property again. This is because of the way the `register` method is written on the `RegistersUsers` trait:

```
public function register(Request $request)
{
    $this->validator($request->all())->validate();

    event(new Registered($user = $this->create($request->all())));

    $this->guard()->login($user);

    return $this->registered($request, $user) ?:
        redirect($this->redirectTo());
}


```

Both create and validator methods consume \$request->all(), however they do so separately, so we have to format each time.

Also, in case you are wondering why we are doing validation on a required field, it's because not all browsers will respect the HTML 5 required tag.

Ok, so once you have all this done, you can register some users and play around with the checkbox options to see your results.

Terms Of Service

While we're here, let's create a route for terms-of-service:

```
Route::get('terms-of-service', 'PagesController@terms');
```

And on the Pages Controller, add the following method:

```
public function terms()
{
    return view('pages.terms-of-service');
}
```

And now a terms-of-service.blade.php in the pages folder in views:

Gist:

[terms-of-service.blade.php](#)

From book:

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

@section('content')

<div class="container">

    <h1>Terms of Service</h1>
    <hr>
    <div><h2>1. Introduction</h2>
    <p>

        Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed
        do eiusmod tempor incididunt ut labore et dolore magna aliqua.
        Ut enim ad minim veniam, quis nostrud exercitation ullamco
        laboris nisi ut aliquip ex ea commodo consequat. Duis aute
        irure dolor in reprehenderit in voluptate velit esse cillum
        dolore eu fugiat nulla pariatur. Excepteur sint occaecat
        cupidatat non proident, sunt in culpa qui officia deserunt
        mollit anim id est laborum.

    </p>
    </div>

    <br><br>
```

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

Privacy

Now that we have our Terms of Service, let's do a privacy page, just to knock it out now. We'll go fast.
in your routes file:

```
Route::get('privacy', 'PagesController@privacy');
```

In PagesController:

```
public function privacy()  
{  
    return view('pages.privacy');  
}
```

Make a view in the pages folder, privacy.blade.php:

Gist:

[privacy.blade.php](#)

The privacy page is too long to be included in the book, please refer to the gist above. The good news is that it is a privacy policy that you can use as a starting point for your own.

Summary

We made a lot of progress in this chapter with our Access Control, but we still have more to go. We have all the building blocks we need to control access in the controller and views, and our application is smart enough to direct the user to the appropriate landing page on login.

Next chapter we will tackle Socialite and get our one-click Facebook login and registration going.

Chapter 8: Socialite - One Click Facebook Login

Ok, we're ready to jump in with our Socialite implementation. I thought it best to do a robust implementation, one worthy of deployment.

So this ends up being one of the tougher builds in the book, we are moving into more intermediate level code. We are dealing with a 3rd party app, Facebook, as well as our environment configuration, making sure the sample-project/auth/facebook/callback is respected properly.

Any little typo in either will cause you grief, and typically, you will not get an error message that is actually helpful. Instead, you end up falling down the rabbit hole of unexpected behavior.

I've done my best to account for this, especially with the custom exceptions we are going to create, which give us a number of breakpoints to hint and diagnose problems.

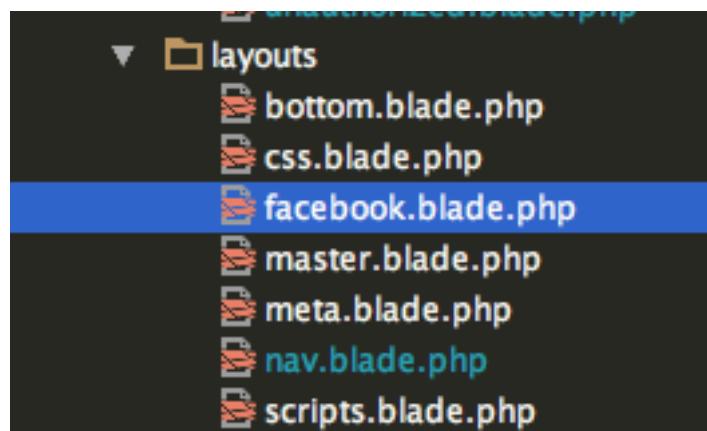
So the point of mentioning this is for you to go through this section carefully with patience. The code I've written has come directly from my IDE in a working project. So if it fails, it's either due to typo or environment problem.

Whatever happens, stick with it, you can get it. It will work. And perhaps all this caution is unnecessary and it will work perfectly the first time you try...

Facebook

We're going to start by setting up a layouts partial to hold our Facebook SDK javascript.

Let's create facebook.blade.php in our layouts folder.



We will leave it empty and come back to it.

Social Routes

Let's add the following routes for our socialite implementation:

```
// socialite routes

Route::get('auth/{provider}', 'Auth\AuthController@redirectToProvider');

Route::get('auth/{provider}/callback',
    'Auth\AuthController@handleProviderCallback');
```

Please note I'm using a structure for the uri that allows us to pass in the name of the provider dynamically, which in our example will be facebook. This will allow you to easily handle other social providers such as github and linkedin.

We will be focusing solely on Facebook in this book, but I will give you hints on how to do more along the way. It will be up to you if you want to build out for other social providers.

Ok, back to our implementation. Let's pull in Socialite via composer.

```
composer require laravel/socialite
```

Just a reminder that doing it this way, as opposed to modifying composer.json, means it will only pull in the package and its dependencies, not go through your entire application and replace whatever has been updated.

Although you should keep your application up-to-date, running composer update can be disruptive. It's up to you how often you feel you should do it.

Next, let's make our app aware Socialite exists. We will modify our config/app.php file in the providers array and add the following line under the Package Service Providers comment:

```
Laravel\\Socialite\\SocialiteServiceProvider::class,
```

And in the same file, in the aliases array, add the following line:

```
'Socialite' => Laravel\\Socialite\\Facades\\Socialite::class,
```

In the config folder, there is also a services.php file. Open that and add the following array:

```
'facebook' => [  
  
    'client_id' => env('FACEBOOK_ID'),  
    'client_secret' => env('FACEBOOK_SECRET'),  
    'redirect' => env('FACEBOOK_URL'),  
  
,
```

So we're setting up the keys we need for our facebook implementation. You can see that we have set the values to the env method, which means this will point to the .env file and look for its values there. So we need to modify that file as well by adding the following:

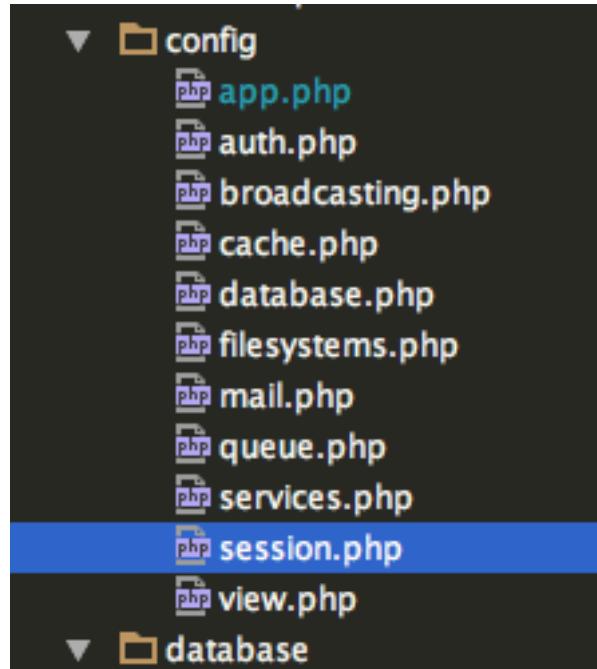
```
FACEBOOK_ID=your-actual-id  
FACEBOOK_SECRET=your-actual-secret  
FACEBOOK_URL=http://sample-project.com/auth/facebook/callback
```

These are just placeholders. We will get the other values later, after we have set up the application. The .env file is one of the really cool features of Laravel that allows you to keep things you might not want to share with other developers via a repository private.

Note you should be using your project url, not mine. Also note, I did not use www, it created an error when I tried that.

session.php

One of the big gotchas in setting up Socialite, and in fact, moving to production for applications for laravel, is a setting in config/session.php located here:



By default that is set to null and we need to make sure that is set to:

```
'domain' => env('SESSION_DOMAIN', 'sample-project.com'),
```

Clear Cookies

You will need to clear the cookie in your browser, once you have made that change. If not, you may get strange behavior because the cookie at that point does not match the session setting. One way to know this is a problem is that when you logout, you get a token mismatch. If that happens, try clearing the cookie, it should solve the problem.

Obviously substitute in your project name for sample-project.

InvalidStateException

I've had some past experiences with the InvalidStateException when moving to deployment, so I thought I would share the solution here:

```
InvalidStateException in AbstractProvider.php line 191:
```

This is an indication there is a problem with either the session.php setting or an environment problem. Check your local host and vhost settings carefully.

You can also run the following commands to clear out cache:

```
php artisan cache:clear  
composer dump-autoload  
php artisan clear-compiled
```

Obviously, you would run those one at a time. If you are still getting the error, look for typos, including in your .env file, and remember to clear your cookies in the browser.

This is one of those things it's wise to remember because it can be a real pain when you are trying to diagnose an environment problem. It would be nice if the error returned more information, but it doesn't give us much. I will try to remind you about these potential issues in the appropriate places.

Tip for nginx users.

A user wrote to me and told me he ran into environment issues, so he had to change his site config to:

```
location / {  
    try_files $uri $uri/ /index.php?$args;  
}
```

Please note that only applies to nginx, and is not a Laravel setting.

Anyway, moving on, we need to setup our Facebook application. I'm going to provide screenshots for this, however, please be aware that there is no way I can reasonably keep up with changes that Facebook might make to the process or to how it looks. This is the fourth time I've written these instructions since 2014, and it's been different every time. So just keep that in mind if you don't see something exactly as I'm displaying it.

That said, you should be able to figure out anything that diverges from what I'm showing without too much effort. The general principles will most likely apply.

Set up Facebook App

Step 1

Go to your [Facebook settings menu](#)

Step 2

Click on [developers link](#) on the bottom of the page.

Step 3

Click on My Apps in the header:



Step 4

From the My Apps dropdown, select Add New App:

[Add a New App](#)

Step 5

New App Modal:

Create a New App ID

Get started integrating Facebook into your app or website

Display Name

Contact Email

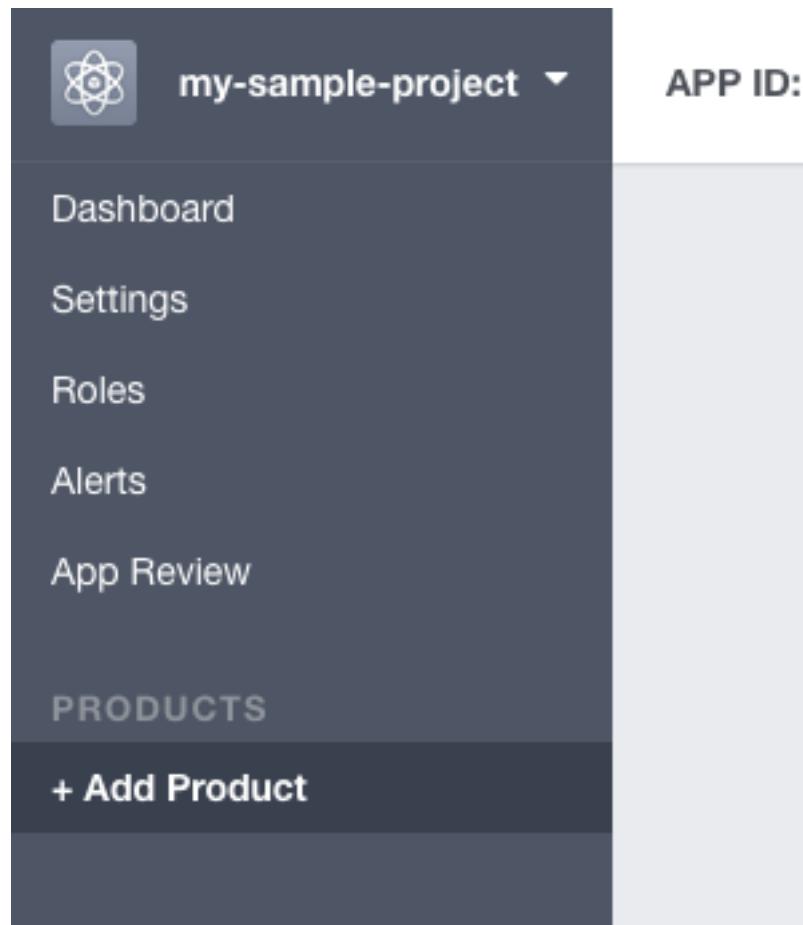
Category

By proceeding, you agree to the [Facebook Platform Policies](#)

Make sure to fill in your email and choose your category.

Step 6

Select Dashboard from admin page:

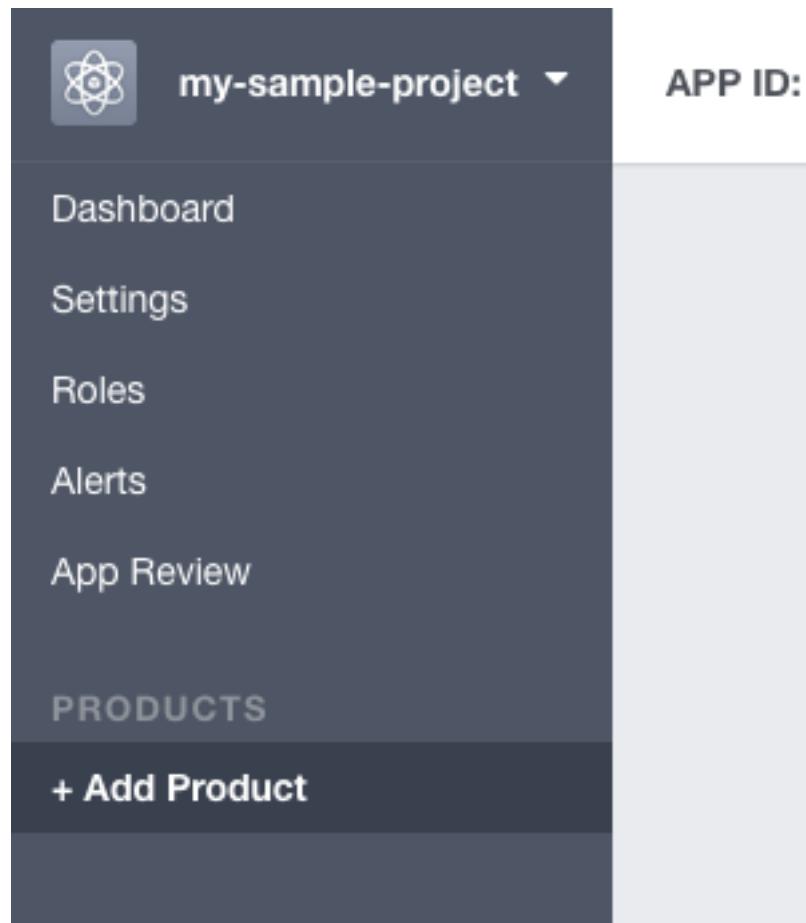


From your dashboard, you can get your app id and your app secret, which you need to put into your FACEBOOK_ID and FACEBOOK_SECRET settings in your .env file.

```
FACEBOOK_ID=your-id
FACEBOOK_SECRET=your-secret
FACEBOOK_URL=http://www.your-project-name.com/auth/facebook/callback
```

Step 7

Click the settings link on the left nav of the Facebook dashboard:



Step 8

You need to add some info to the basic settings:

App ID	App Secret
1435542843156693	***** Show
Display Name	Namespace
sample-project	
App Domains	Contact Email
	your@email.com
Privacy Policy URL	Terms of Service URL
Privacy policy for Login dialog and App Details	Terms of Service for Login dialog and App Details
App Icon	Category
 1024 x 1024	Education ▾

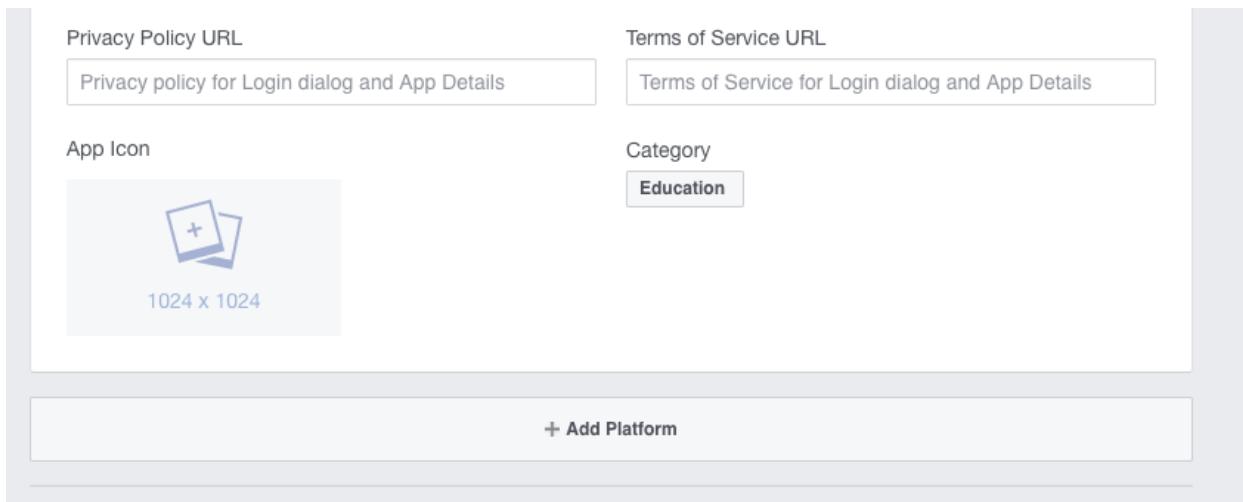
Don't worry about namespace, but all others should have a value. In App Domains, I'm putting sample-project.com. Save your changes. It will show an error because we have not yet selected our platform, but that's ok, we are about to do that step.

Note: if your project is ending in something other than .com, such as .dev or .project, you may experience problems getting this to work, and the error messages won't be particularly helpful. I recommend using the .com extension when setting up your facebook login.

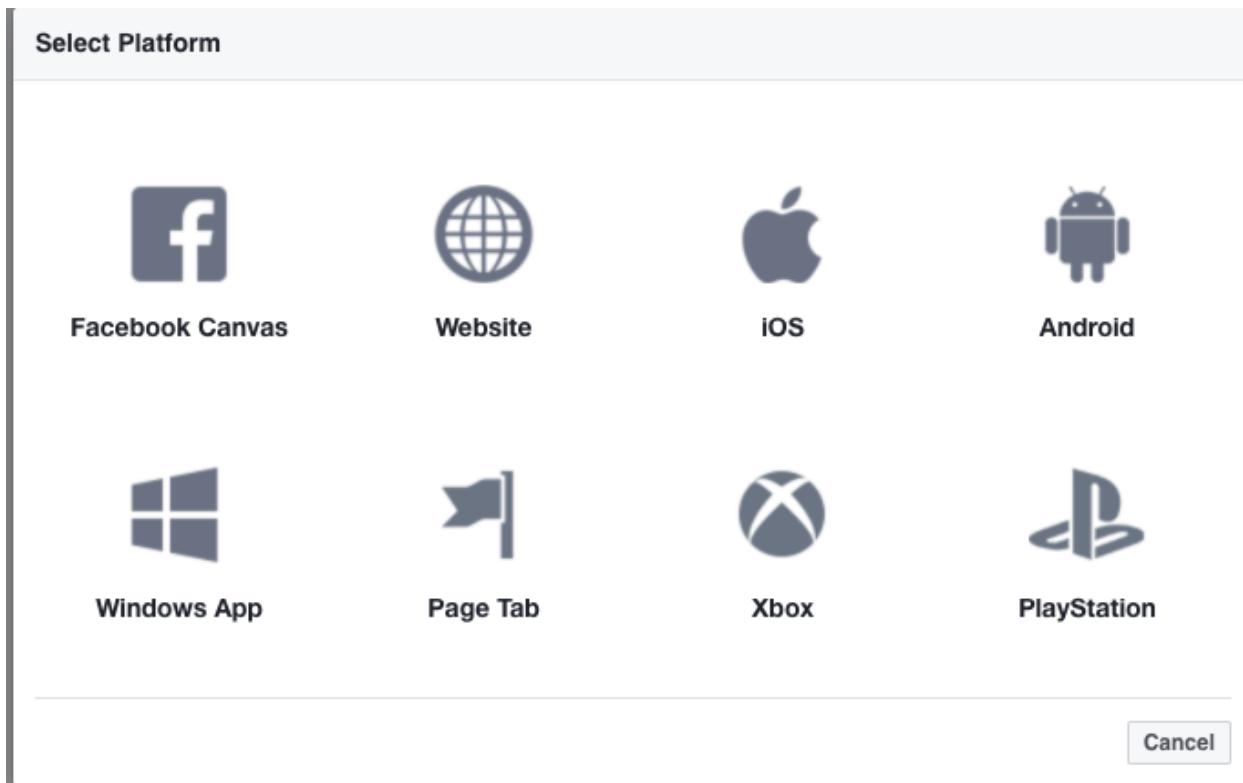
Also, if you are working in a dev environment, it might throw an error for privacy policy and terms of service urls. If that is the case, leave it blank for now. You can add them later when you are in production.

Step 9

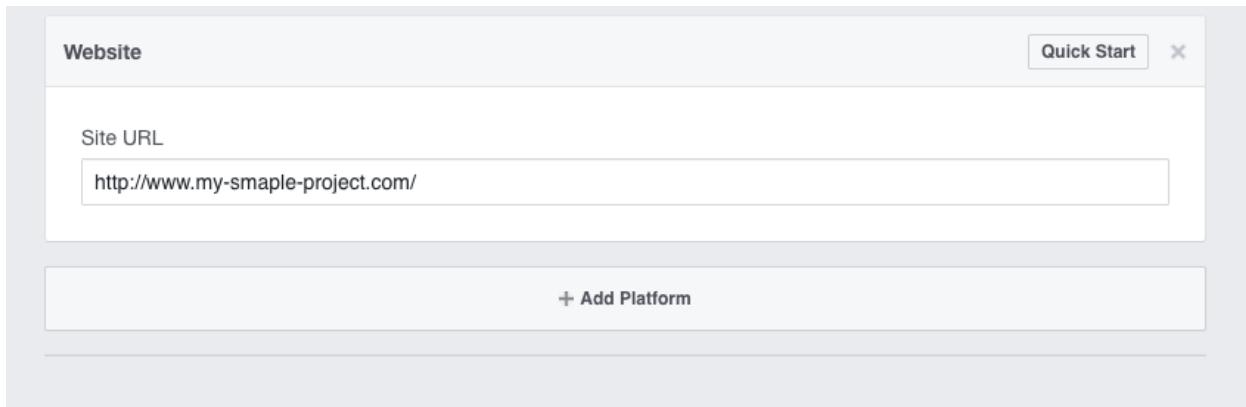
Click Add Platform button:



Select website:



Then enter your URL:



Save changes. Then click on the Quick Start link in the top right of the image above.

Step 10

The Facebook SDK will open on a new tab. We get the following code:

```
<script>

  window.fbAsyncInit = function() {
    FB.init({
      appId      : 'your-app-id',
      xfbml      : true,
      version    : 'v2.7'
    });
  };

  (function(d, s, id){
    var js, fjs = d.getElementsByTagName(s)[0];
    if (d.getElementById(id)) {return;}
    js = d.createElement(s); js.id = id;
    js.src = "//connect.facebook.net/en_US/sdk.js";
    fjs.parentNode.insertBefore(js, fjs);
  })(document, 'script', 'facebook-jssdk'));

</script>
```

Note, I have cut out my appId and replaced it with ‘your-app-id’, which obviously you should contain your actual id. In fact you should just copy the code directly from facebook, not from here.

The instructions say to put the code immediately after our opening body tag. So we will copy this code to our facebook.blade.php file, then call it in our master.blade.php file like so:

```
<body role="document">  
    @include('layouts.facebook')  
    @include('layouts.nav')
```

The line above and below are given just for reference.

Next you need to enter your domain. This domain should match your host entry, otherwise Facebook might not allow it work:

sample-project.com

The next step is to test our Facebook integration.

We can pop this into pages.index page to see if it works. We’ll put in between the breadcrumb and jumbotron, like so:

```
<ol class="breadcrumb">  
    <li><a href="#">Home</a></li>  
</ol>  
  
<div  
    class="fb-like"  
    data-share="true"  
    data-width="450"  
    data-show-faces="true">  
</div>  
  
<!-- Main jumbotron for a primary marketing message or call to action -->  
<div class="jumbotron">
```

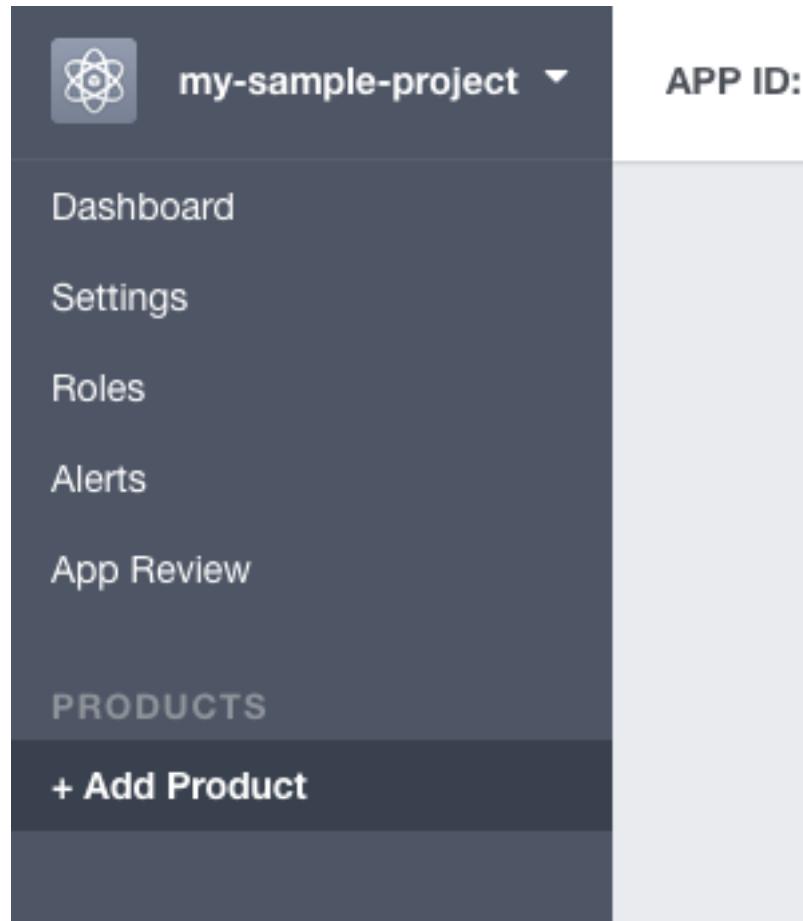
Then you should be able to see it and test it like so:

The screenshot shows a website titled "Sample Project". At the top, there is a dark header bar with the title "Sample Project" on the left and navigation links "Home", "About", "Content", and "gumby" on the right, along with a user profile icon. Below the header, a light gray sidebar on the left contains a "Home" link. The main content area features a large heading "Sample Project" and a subtext "Use it as a starting point to create something more unique by building on or modifying it.". At the bottom of the main content area, there is a copyright notice: "© 2015 - 2017 Sample Project All rights Reserved." Above the main content area, there is a small social sharing section with "Like" and "Share" buttons and a note that "One person likes this. Be the first of your friends."

So now you have verification that your Facebook connection is working. However, we still have work to do.

Step 11

The next step is to add the login service. Click add Product link:



We are going to select the following product:

A screenshot of a web-based product setup interface. The title "Product Setup" is at the top. Below it is a section for "Facebook Login" which includes the text "The world's number one social login product." and a "Get Started" button.

Click on get started. You will get the following:

The screenshot shows the 'Client OAuth Settings' section of a web application. It contains several configuration options with checkboxes:

- Client OAuth Login:** A checked 'Yes' button. Description: Enables the standard OAuth client token flow. Secure your application and prevent abuse by locking down which token redirect URLs are allowed with the options below. Disable globally if not used. [?]
- Web OAuth Login:** An unchecked 'No' button. Description: Enables web based OAuth client login for building custom login flows. [?]
- Force Web OAuth Reauthentication:** An unchecked 'No' button. Description: When on, prompts people to enter their Facebook password in order to log in on the web. [?]
- Embedded Browser OAuth Login:** An unchecked 'No' button. Description: Enables browser control redirect uri for OAuth client login. [?]
- Valid OAuth redirect URIs:** A text input field containing `http://sample-project.com/auth/facebook/callback`.
- Login from Devices:** An unchecked 'No' button. Description: Enables the OAuth client login flow for devices like a smart TV [?]

At the bottom left is a 'Deauthorize' button, and at the bottom right are 'Discard' and 'Save Changes' buttons.

Enter your callback URI into the Valid OAuth redirect URIs field and save.

For my project, the callback uri is:

`http://sample-project.com/auth/facebook/callback`

And that's pretty much it for the facebook side of the implementation.

Again note that I did not use the www prefix, that caused an error for me.

You also have an App Review link on the admin page. Just mentioning it here, so if you take your app live, you need to go to that page and you will need to slide over to yes.

The screenshot shows three main sections of the Facebook App Review interface:

- Make my-sample-project public?**: A button labeled "Yes" is highlighted, with the message "Your app is currently **live** and available to the public." displayed.
- Submit Items for Approval**: A message states that some integrations require approval before public usage, and links to the **Platform Policy** and **Review Guidelines**. A "Start a Submission" button is present.
- Approved Items [?]**: A list of approved login permissions:
 - email** [?]: Provides access to the person's primary email address. This permission is approved by default.
 - public_profile** [?]: Provides access to a person's basic information, including first name, last name, profile picture, gender and age range. This permission is approved by default.
 - user_friends** [?]: Provides access to a person's list of friends that also use your app. This permission is approved by default.

Navigation To Facebook

So let's add a link to allow us to join or login with Facebook. We can the following li in nav.blade.php, inside of the else statement that checks for logged in or not via Auth::check(). I will show the entire else statement for reference, but you only need to add one :

```
@else

<li><a href="/login">Login</a></li>
<li><a href="/register">Register</a></li>
<li>
    <a href="/auth/facebook">
        <i class="fa fa-facebook"></i>
        &nbsp;&nbsp;
        Sign in
    </a>
</li>

@endif
```

Now let's add a link nav.blade.php in the username drop down on the nav as well, put it above the Logout :

```
<li>
    <a href="/auth/facebook">
        <i class="fa fa-facebook"></i>
        &nbsp;&nbsp;
        Sync
    </a>
</li>
```

It's the same url as our login because our internal logic determines what to do with it. For reference, here is the entire nav file:

Gist:

nav.blade.php

We don't have a route yet for that, so it won't work, but we will make that at a future point.

Next, let's create a data structure that will let us add more than one social provider per user. Let's run the following from the command line:

```
php artisan make:model SocialProvider -m
```

So that will give us a migration stub, which we can add to, so it looks like this:

Gist:

Social Provider Migration

From book:

```
<?php

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

class CreateSocialProvidersTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */

    public function up()
    {
        Schema::create('social_providers', function (Blueprint $table) {

            $table->increments('id');
            $table->integer('user_id')->unsigned()->index();
            $table->string('source');
            $table->string('source_id')->unique()->index();
            $table->string('avatar');
            $table->timestamps();

        });
    }
}
```

```
}

/**
 * Reverse the migrations.
 *
 * @return void
 */

public function down()
{
    Schema::dropIfExists('social_providers');

}
}
```

Then run the migration:

```
php artisan migrate
```

Let's add a protected \$fillable property and the relationship User on the SocialProvider model:

Gist:

[SocialProvider.php](#)

From book:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class SocialProvider extends Model
{

    protected $fillable = ['user_id',
        'source',
        'access_token',
        'refresh_token',
        'expires_at'];
}
```

```
'source_id',
'avatar'];

public function user()
{
    return $this->belongsTo('App\User');

}
```

We are using a belongsTo because Users may have more than one SocialProvider record. We also need to set up the inverse, so let's add ahasMany on the User.php model:

```
public function socialProviders()
{
    return $this->hasMany('App\SocialProvider');

}
```

Notice the plural convention on the method name. This is how we name relationship methods that have more than one word.

Integrating Socialite

We are going to integrate our social login and registration into a controller, which we are going to build shortly.

We want users to be able to register or login with a single click. They will of course have to click to approve the application, but that is on the provider side and there is nothing I can do about that. After they do that once, it's a one click sign in.

If someone has an existing account with our application, and they try logging in with Facebook, and their email address is the same as what is in their account, they will have to sync their accounts from within the application, which means they will have to login first.

So there are a number of complicated scenarios that we have to account for. And then of course there could be unexpected problems. Our common unique identifier will be the email address, but what happens if the social provider doesn't return one?

I haven't run into that problem with Facebook, but I have with other social networks, so anticipating these kinds of issues goes a long ways toward making our app extensible, even though for now, we are only implementing Facebook.

Exceptions

One of the things I did with this implementation that I found helpful was to create a bunch of custom exceptions that would help me troubleshoot problems. So we're going to start by creating 5 new exceptions:

- EmailAlreadyInSystemException
- AlreadySyncedException
- CredentialsDoNotMatchException
- ConnectionNotAcceptedException
- TransactionFailedException

Ok, start by creating an exception for each of the above in the Exceptions Folder. You should know how to make these. I will provide Gists, but we will not review since this is identical to what we have done before.

[EmailAlreadyInSystemException](#)

[AlreadySyncedException](#)

[CredentialsDoNotMatchException](#)

[ConnectionNotAcceptedException](#)

[TransactionFailedException](#)

Note: Be careful about typos when naming the files.

Here is the Gist for the revised Handler.php file:

[Handler.php](#)

You'll note that the case statements are ordered alphabetically by exception type, which makes it easier to find them if you have to troubleshoot.

Now we'll do the same and create the views in the errors folder:

[already-synced-exception.blade.php](#)

[connection-not-accepted-exception.blade.php](#)

[credentials-do-not-match-exception.blade.php](#)

[email-already-in-system-exception.blade.php](#)

[email-not-provided-exception.blade.php](#)

transaction-failed-exception.blade.php

Make sure you used the .blade.php convention.

Ok, glad we got that out of the way.

A Big Heads Up

If you make a mistake in your Handler.php file, you can have big problems, so be careful when working with this file.

I had a typo that knocked out error reporting for my application. I just copied and pasted right in the middle of:

default:

```
return parent:://my-mistake-was-here//render($request, $e);
```

It was goodnight Irene after that. 2 hours of research to figure out why I wasn't getting error reporting. I looked around, and there were useful comments about writing permissions to your storage folder. But that was not the problem. Eventually I swapped out line by line in Handler.php until I tripped over the problem.

So I'm giving you the benefit of my pain. Tread carefully with the Handler.php file.

Let's also modify our '/' route to the following:

```
Route::get('/', 'PagesController@index')->name('home');
```

Naming the route will make it easier to work with. We will need it in our redirectUser method that we will be using. We will get to that in a few minutes.

AuthController

Ok, moving on. Let's make a controller named AuthController. It will combine methods from both the RegisterController and the LoginController, which we will need.

Let's create it with the following command:

```
php artisan make:controller Auth/AuthController
```

The Auth/ prefix will place the new controller in the Auth folder.

Tip For Staying Current

Before we update that file, let me just point out that the login method on `AuthenticatesUsers` tends to change with subsequent releases of Laravel, of which there are many. If you run `composer update`, you may be updating your traits, so be sure to check for that.

To check your version of laravel, run the following from the command line:

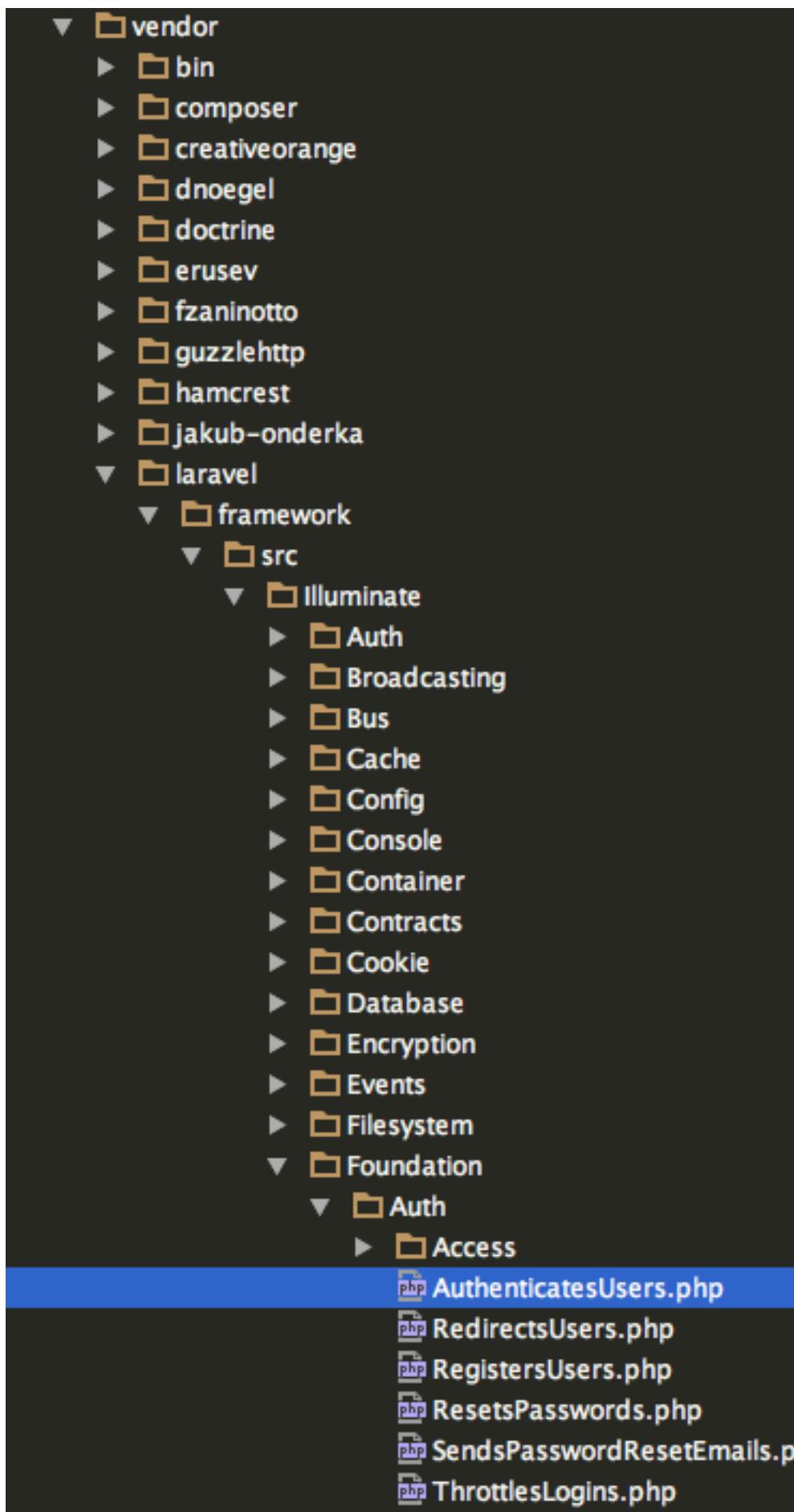
```
php artisan -V
```

Laravel is constantly releasing minor versions. It's not a bad thing, they keep improving it, but it is something you need to be aware of, incase a release comes out after this version of the book is published, and I have not had a chance to update the book.

In our `AuthController`, as we will see in a minute, we overwrite the existing login method by placing it directly into the `AuthController`. It is meant to be exactly the same method as the one on the `AuthenticatesUsers` trait.

So, in the course of doing this, if you see that they are not the same as what's in the book, then to stay up-to-date, modify the login method in `AuthController` to match that of the one in `AuthenticatesUsers`.

We are current now, so you shouldn't have to worry about it, but it doesn't hurt to double-check it either. You can find `AuthenticatesUsers` at:



If the image isn't clear, the path is:

vendor/laravel/framework/src/Illuminate/Foundation/Auth

Ok, let's go ahead and change the contents of AuthController.php to the following:

Gist:

[AuthController.php](#)

From book:

```
<?php

namespace App\Http\Controllers\Auth;

use App\Exceptions\NoActiveAccountException;
use App\Http\Traits\Social\ManagesSocialAuth;
use Illuminate\Support\Facades\Auth;
use Illuminate\Http\Request;
use Illuminate\Foundation\Auth\AuthenticatesUsers;
use App\Http\Requests;
use Socialite;

class AuthController extends RegisterController
{
    use AuthenticatesUsers, ManagesSocialAuth;

    protected $redirectTo = '/';

    /**
     * Create a new controller instance.
     *
     * @return void
     */

    public function __construct()
    {
        $this->middleware('guest', [
            'except' => ['logout',

```

```
'redirectToProvider',
'handleProviderCallback']

]);


}

private function checkStatusLevel()
{
    if ( ! Auth::user()->isActiveStatus()) {

        Auth::logout();

        throw new NoActiveAccountException;

    }

}

public function redirectPath()
{
    if (Auth::user()->isAdmin()){

        return 'admin';

    }

    return property_exists($this, 'redirectTo') ? $this->redirectTo : '/';

}

/**
 * Handle a login request to the application.
 *
 * @param \Illuminate\Http\Request $request
 * @return \Illuminate\Http\Response
 */

public function login(Request $request)
{
    $this->validateLogin($request);
```

```
// If the class is using the ThrottlesLogins trait,  
// we can automatically throttle  
// the login attempts for this application.  
// We'll key this by the username and  
// the IP address of the client making these  
// requests into this application.  
  
if ($this->hasTooManyLoginAttempts($request)) {  
  
    $this->fireLockoutEvent($request);  
  
    return $this->sendLockoutResponse($request);  
  
}  
  
if ($this->attemptLogin($request)) {  
  
    $this->checkStatusLevel();  
  
    return $this->sendLoginResponse($request);  
}  
  
// If the login attempt was unsuccessful we will  
// increment the number of attempts  
// to login and redirect the user back to the  
// login form. Of course, when this  
// user surpasses their maximum number of attempts  
// they will get locked out.  
  
$this->incrementLoginAttempts($request);  
  
return $this->sendFailedLoginResponse($request);  
  
}  
  
}
```

You'll notice that looks a lot like our LoginController, so much so, that it's actually code duplication, which we don't want. Before we fix that, let's learn a little about the AuthController.

The AuthController has to be responsible for both registration and login of new social users and at the same time, we want to leverage all the strengths of the existing classes and methods that Laravel provides us out of the box.

So the first item of interest:

```
class AuthController extends RegisterController
{
    use AuthenticatesUsers, ManagesSocialAuth;
```

You can see we are extending the RegisterController and are using the AuthenticatesUsers trait, which is used by the LoginController. We are also calling in the ManagesSocialAuth trait, which we have not made yet.

So at this point the AuthController is capable of doing everything that our Login and Register Controllers are doing, so we don't need the duplicated methods that are on the LoginController.

Let's remove the duplicate methods from the LoginController, so that it looks like this:

Gist:

[LoginController](#)

From book:

```
<?php
```

```
namespace App\Http\Controllers\Auth;

use App\Http\Controllers\Controller;
use Illuminate\Foundation\Auth\AuthenticatesUsers;

class LoginController extends Controller
{
    /*
    |--------------------------------------------------------------------------
    | Login Controller
    |--------------------------------------------------------------------------
    |
    | This controller handles authenticating users for the application and
    | redirecting them to your home screen. The controller uses a trait
    | to conveniently provide its functionality to your applications.
    |
    */
    use AuthenticatesUsers;

    /**

```

```
* Where to redirect users after login / registration.  
*  
* @var string  
*/  
  
protected $redirectTo = '/';  
  
/**  
 * Create a new controller instance.  
 *  
 * @return void  
 */  
  
public function __construct()  
{  
  
    $this->middleware('guest', ['except' => 'logout']);  
  
}  
  
}
```

That should look a lot like it did right out of the box. One of our goals is to keep the footprint on the code supplied by the framework as light as possible.

The reason for this is that when Laravel comes out with a new feature for login or registration, we want to be able to leverage that into our application. When they do a version change, we will probably have to update our code if we want to use it, but the changes will be minimal and easy to maintain.

Don't test login and registration just yet, they are not going to work for two reasons.

The register and login routes are pointing to the wrong controller. We are calling a trait that doesn't exist yet.

So the first problem is easy enough to fix. We need to drop the Router::auth() method from the routes file and replace it with auth routes that are pointing to the correct controller. This includes the forgot password routes.

I will give you the entire routes file for reference:

Gist:

[web.php](#)

From book:

```
<?php

/*
|--------------------------------------------------------------------------
| Web Routes
|--------------------------------------------------------------------------
|
| Here is where you can register web routes for your application. These
| routes are loaded by the RouteServiceProvider within a group which
| contains the "web" middleware group. Now create something great!
|
*/

// Admin route

Route::get('/admin', 'AdminController@index')->name('admin');

// Authentication routes

Route::get('login', 'Auth\AuthController@showLoginForm')
    ->name('login');

Route::post('login', 'Auth\AuthController@login');

Route::post('logout', 'Auth\AuthController@logout')
    ->name('logout');

// home page route

Route::get('/', 'PagesController@index')->name('home');

// Password routes

// Password Reset Routes...

Route::get('password/reset',
    'Auth\ForgotPasswordController@showLinkRequestForm')
    ->name('password.request');

Route::post('password/email',
    'Auth\ForgotPasswordController@sendResetLinkEmail');

Route::get('password/reset/{token}',
```

```
'Auth\ResetPasswordController@showResetForm')
->name('password.reset');

Route::post('password/reset', 'Auth\ResetPasswordController@reset');

// Privacy route

Route::get('privacy', 'PagesController@privacy');

// Registration routes

Route::get('register',
    'Auth\AuthController@showRegistrationForm')->name('register');

Route::post('register', 'Auth\AuthController@register');

// Socialite routes

Route::get('auth/{provider}', 'Auth\AuthController@redirectToProvider');

Route::get('auth/{provider}/callback',
    'Auth\AuthController@handleProviderCallback');

// Terms route

Route::get('terms-of-service', 'PagesController@terms');

// test route

Route::get('test', 'TestController@index')
->middleware(['auth', 'throttle']);

// Widget routes

Route::get('widget/create',
    'WidgetController@create')->name('widget.create');

Route::get('widget/{id}-{slug?}',
    'WidgetController@show')->name('widget.show');

Route::resource('widget',
    'WidgetController', ['except' => ['show', 'create']]);

```

As a reminder, use the gist for formatting, your code will be more readable.

You can see we dropped the Route::auth() method and added the login, register, and password routes:

```
// Authentication routes

Route::get('login', 'Auth\AuthController@showLoginForm')
    ->name('login');

Route::post('login', 'Auth\AuthController@login');

Route::post('logout', 'Auth\AuthController@logout')
    ->name('logout');

// Registration routes

Route::get('register',
    'Auth\AuthController@showRegistrationForm')->name('register');

Route::post('register', 'Auth\AuthController@register');

// Password Reset Routes...

Route::get('password/reset',
    'Auth\ForgotPasswordController@showLinkRequestForm')
    ->name('password.request');

Route::post('password/email',
    'Auth\ForgotPasswordController@sendResetLinkEmail');

Route::get('password/reset/{token}',
    'Auth\ResetPasswordController@showResetForm')
    ->name('password.reset');

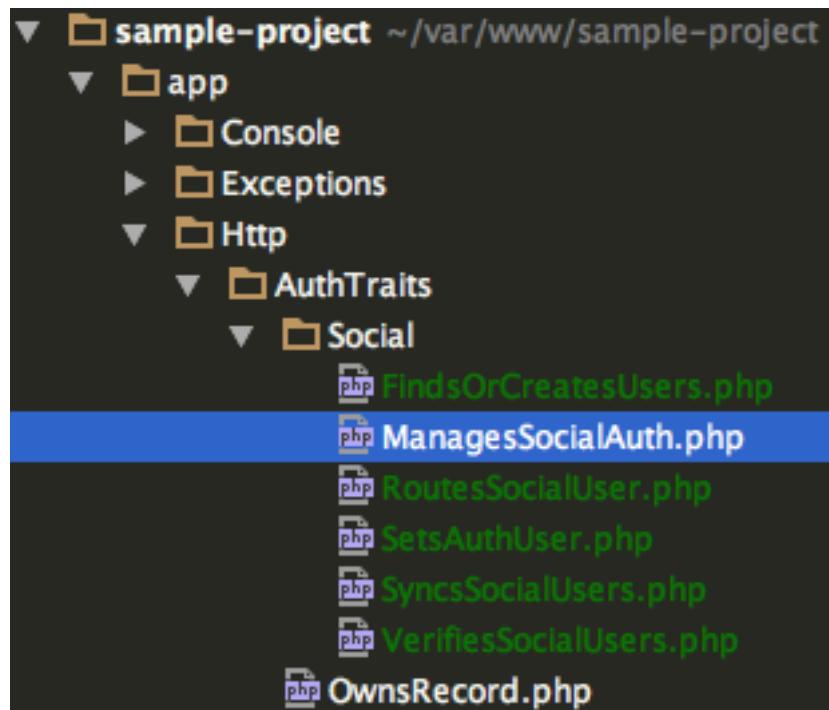
Route::post('password/reset', 'Auth\ResetPasswordController@reset');
```

Getting back to the AuthController, at this point, we only see a few methods from the LoginController that we have already covered. Since we are extending the RegisterController, we already know the methods it's

extending. This means we are getting a lot of mileage out of work we have already done. So all we need to do now is handle the social authentication.

ManagesSocial.php

We are going to create a trait that will not be used by other classes, other than the AuthController. However, it will help us keep the size of AuthController more manageable and readable. It will be located here:



Don't worry about the other files in the Social folder, we will create them in a few minutes.

First, you need to create a Social folder inside your AuthTraits folder. This will help us keep all these traits separated.

Ok, let's dig in. In your AuthTraits folder, create ManagesSocialAuth.php with the following contents:

Gist:

[ManagesSocialAuth.php](#)

From book:

```
<?php

namespace App\Http\Auth\Traits\Social;

use App\Exceptions\EmailNotProvidedException;
use Redirect;
use Socialite;

trait ManagesSocialAuth
{

    // the traits contain the methods needed
    // for the handleProviderCallback

    use FindsOrCreatesUsers,
        RoutesSocialUser,
        SetsAuthUser,
        SyncsSocialUsers,
        VerifiesSocialUsers;

    private $provider;

    private $approvedProviders = [ 'facebook', 'github'];

    public function handleProviderCallback($provider)
    {

        $this->verifyProvider($this->provider = $provider);

        $socialUser = $this->getUserFromSocialite($provider);

        $providerEmail = $socialUser->getEmail();

        if ($this->socialUserHasNoEmail($providerEmail)) {

            throw new EmailNotProvidedException;

        }

        if ($this->socialUserAlreadyLoggedIn()) {

            $this->checkIfAccountSyncedOrSync($socialUser);
```

```
}

// set authUser from socialUser

$authUser = $this->setAuthUser($socialUser);

$this->loginAuthUser($authUser);

$this->logoutIfUserNotActiveStatus();

return $this->redirectUser();

}

}
```

If you are copying, please remember to use the gist.

You can see we pull in all the other traits we are going to create:

```
use FindsOrCreatesUsers,
RoutesSocialUser,
SetsAuthUser,
SyncsSocialUsers,
VerifiesSocialUsers;
```

What we are trying to accomplish is to keep our code separated into logical components, and the trait names reflect this.

Using nested traits like this is just simply moving out methods to their own traits so you don't have a giant size file.

Since these traits are being called by ManagesSocialAuth, which is itself called by the AuthController, another option is to put all the methods in the controller, it is effectively the same thing in this case.

But I think you'll agree that breaking it down into smaller more digestible units is a better way to organize the code. At least I hope so. The one tradeoff is that you have follow the methods into the various traits when you are working on them.

Many IDEs support a shortcut key to finding the origin of the method. In PHP Storm, it's command click, if you hover over the method name. That will take you to the method itself, which means you don't have to memorize every method location. I'm mentioning that for the sake of anyone who might not be following that practice.

Ok, let's get the other traits made. We will start with `FindsOrCreatesUsers`:

[FindsOrCreatesUsers.php](#)

From book:

```
<?php

namespace App\Http\Auth\Traits\Social;

use App\User;
use App\SocialProvider;
use App\Exceptions\EmailAlreadyInSystemException;
use App\Exceptions\CredentialsDoNotMatchException;
use Illuminate\Support\Facades\DB;
use App\Exceptions\TransactionFailedException;

trait FindsOrCreatesUsers
{
    /**
     * Return user if exists; create and return if not
     *
     * @param $facebookUser
     * @return User
     */

    private function findOrCreateUser($socialUser)
    {
        // is email already in table?

        if ( $authUser = $this->userWhereEmailMatches($socialUser)){

            // scenario where email is already in table
            // is the provider source correct and does the
            // social id match?
        }
    }
}
```

```
// if there is a match, return $authUser,  
// if not throw exception  
  
if ( ! $this->matchesIds($socialUser, $authUser)) {  
  
    // exception instructs user to login first, then sync  
    // covers scenario where there are matching emails,  
    // but no matching id  
    // or where there is existing social id and it  
    // doesn't match  
  
    throw new EmailAlreadyInSystemException;  
  
}  
  
// if email and id matches, return the $authUser  
  
return $authUser;  
  
}  
  
// scenario where no matching email,  
// but social id already exists  
  
if ($this->socialIdAlreadyExists($socialUser)){  
  
    throw new CredentialsDoNotMatchException;  
  
}  
  
// if no user matching social exists, we create one  
  
$authUser = $this->makeNewUser($socialUser);  
  
return $authUser;  
  
}  
  
private function makeNewUser($socialUser)  
{
```

```
//create user if not already exists and email does not
// already exist.

$password = $this->makePassword();

DB::beginTransaction();

try{

    $authUser = User::create([
        'name' => $socialUser->name,
        'email' => $socialUser->email,
        'password' => $password,
        'status_id' => 10,
    ]);

    SocialProvider::create([
        'user_id' => $authUser->id,
        'source' => $this->provider,
        'source_id' => $socialUser->id,
        'avatar' => $socialUser->avatar,
    ]);

    DB::commit();

} catch (\Exception $e) {

    DB::rollback();

    throw new TransactionFailedException();

}

return $authUser;

}
```

```
/**  
 * @return string  
 */  
  
private function makePassword()  
{  
    $passwordParts = rand() . str_random(12);  
    $password = bcrypt($passwordParts);  
  
    return $password;  
}  
  
}
```

I tried to name the trait intuitively. We find the user or create a new one. We will explore this in detail in a couple of minutes.

Ok, next trait is RoutesSocialUser:

Gist:

[RoutesSocialUser.php](#)

From book:

```
<?php  
  
namespace App\Http\Auth\Traits\Social;  
  
use Socialite;  
use Illuminate\Support\Facades\Auth;  
use App\Exceptions\ConnectionNotAcceptedException;  
  
trait RoutesSocialUser  
{  
  
    /**  
     * @param $authUser  
     */
```

```
private function loginAuthUser($authUser)
{
    Auth::login($authUser, true);

}

private function logoutIfUserNotActiveStatus()
{
    return $this->checkStatusLevel();
}

public function redirectToProvider($provider)
{
    return Socialite::driver($provider)->redirect();
//->scopes(['public_profile', 'email'])->redirect();

}

private function redirectUser()
{
    if (Auth::user()->isAdmin()){

        return redirect()->route('admin');

    }

    return redirect()->route('home');

}

private function getUserFromSocialite($provider)
{
    try {

        $socialUser = Socialite::driver($provider)->user();

        return $socialUser;

    } catch (\Exception $e) {

```

```
        throw new ConnectionNotAcceptedException;

    }

}
```

This trait holds the methods that move the user around. We will go through these methods as they are called.

Next, we need to make SetsAuthUser.php:

Gist:

[SetsAuthUser.php](#)

From book:

```
<?php

namespace App\Http\Auth\Traits\Social;

trait SetsAuthUser
{

    /**
     * @param $socialUser
     * @return User
     * @throws \App\Exceptions\CredentialsDoNotMatchException
     * @throws \App\Exceptions\EmailAlreadyInSystemException
     */

    private function setAuthUser($socialUser)
    {

        $authUser = $this->findOrCreateUser($socialUser);

        return $authUser;

    }

}
```

Just one method on this trait to set the authenticated user. It will make more sense in context, when we see it in action.

Ok, next let's make SyncsSocialUsers.php:

Gist:

[SyncsSocialUsers.php](#)

From book:

```
<?php

namespace App\Http\Auth\Traits\Social;

use App\Exceptions\AlreadySyncedException;
use Illuminate\Support\Facades\Auth;
use App\Http\Requests;
use App\SocialProvider;
use App\Exceptions\CredentialsDoNotMatchException;

trait SyncsSocialUsers
{

    /**
     * @param $facebookUser
     * @return mixed
     */

    private function accountSynced($socialUser)
    {
        if ($this->authUserEmailMatches($socialUser)){
            return $this->verifyUserIds($socialUser);
        }

        return false;
    }
}
```

```
private function checkIfAccountSyncedOrSync($socialUser)
{
    //if you are logged in and accountSynced is true,
    // you are already synced

    if ($this->accountSynced($socialUser)){
        throw new AlreadySyncedException;

    } else {

        // check for email match

        if ( ! $this->authUserEmailMatches($socialUser)) {
            throw new CredentialsDoNotMatchException;
        }

        // if emails match, then sync accounts

        $this->syncUserAccountWithSocialData($socialUser);

        alert()->success('Confirmed!', 'You are now synced...');

        return $this->redirectUser();
    }
}

private function syncUserAccountWithSocialData($socialUser)
{
    // one last check to see if the social id already exists

    if ($this->socialIdAlreadyExists($socialUser)){
        throw new CredentialsDoNotMatchException;
    }
}
```

```
// lookup user id and update create provider record

SocialProvider::create([
    'user_id' => Auth::user()->id,
    'source'   => $this->provider,
    'source_id' => $socialUser->id,
    'avatar'   => $socialUser->avatar
]);

}
```

This trait holds the methods responsible for syncing an existing user account to a social account.

And finally, let's make VerifiesSocialUsers.php:

Gist:

[VerifiesSocialUsers.php](#)

From book:

```
<?php

namespace App\Http\Auth\Traits\Social;

use App\Exceptions\UnauthorizedException;
use Illuminate\Support\Facades\Auth;
use App\SocialProvider;
use App\User;

trait VerifiesSocialUsers
{
    private function authUserEmailMatches($socialUser)
    {

```

```
    return $socialUser->email == Auth::user()->email;

}

/** 
 * @param $facebookUser
 * @return mixed
 */

private function socialIdAlreadyExists($socialUser)
{
    return SocialProvider::where('source_id', '=', $socialUser->id)->exists();
}

/** 
 * @return mixed
 */

private function socialUserAlreadyLoggedIn()
{
    return Auth::check();
}

private function socialUserHasNoEmail($socialUserEmail)
{
    return $socialUserEmail == null;
}

private function verifyProvider($provider)
{
    if ( ! in_array($provider, $this->approvedProviders)){
        throw new UnauthorizedException();
    }
}
```

```
}

/**
 * @param $socialUser
 * @return bool
 */

private function verifyUserIds($socialUser)
{
    return (SocialProvider::where('source_id', $socialUser->id)
        ->where('user_id', Auth::id())
        ->where('source', $this->provider)
        ->exists()) ? true : false;
}

/**
 * @param $socialUser
 * @return mixed
 */

private function userWhereEmailMatches($socialUser)
{
    return User::where('email', $socialUser->email)->first();
}

/**
 * @param $socialUser
 * @param $authUser
 * @return mixed
 */

private function matchesIds($socialUser, User $authUser)
{
    return $authUser->socialProviders()
        ->where('source', $this->provider)
        ->where('source_id', $socialUser->id)
        ->first();
}
```

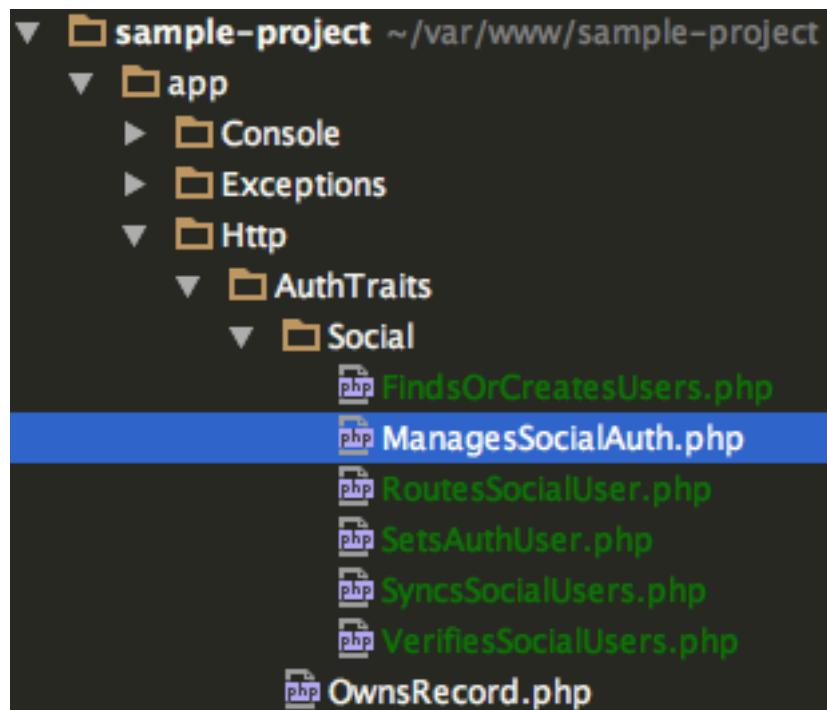
```
}
```

```
}
```

This trait has methods that do simple checks to see if we are getting the correct user and that we have what we need to continue.

By themselves the methods in the traits seem disconnected, but you will see how it all fits together soon.

Just to give the folder another look, it should contain the following:



Putting It All Together

When implementing Socialite, we reference two basic methods from the [Socialite docs](#):

```
/*
 * Redirect the user to the GitHub authentication page.
 *
 * @return Response
 */

public function redirectToProvider()
{
    return Socialite::driver('github')->redirect();
}

/**
 * Obtain the user information from GitHub.
 *
 * @return Response
 */

public function handleProviderCallback()
{
    $user = Socialite::driver('github')->user();
    // $user->token;
}
```

In this case they are using Github as an example, and obviously, we will be using facebook.

Our redirectToProvider method is virtually identical to the example, except that we are including an argument to the method, the \$provider:

```
public function redirectToProvider($provider)
{
    return Socialite::driver($provider)->redirect();
    //->scopes(['public_profile', 'email'])->redirect();
}
```

If you recall, we have a route that makes this call:

```
Route::get('auth/{provider}', 'Auth\AuthController@redirectToProvider');
```

So that points to the AuthController, which pulls in ManagesSocialAuth, which pulls in RoutesSocialUsers, where we find our redirectToProvider method:

```
public function redirectToProvider($provider)
{
    return Socialite::driver($provider)->redirect();
    //->scopes(['public_profile', 'email'])->redirect();
}
```

It's literally the same method from the docs.

So what happens is the outbound signal gets sent to the provider that we passed along in the route, and then the provider returns a callback, which hits the following route:

```
Route::get('auth/{provider}/callback',
'Auth\AuthController@handleProviderCallback');
```

This returns to the handleProviderCallback method, which the AuthController pulls in via the ManagesSocialAuth trait. Obviously we modified the handleProviderCallback method, which does all the heavy lifting.

```
public function handleProviderCallback($provider)
{
    $this->verifyProvider($this->provider = $provider);

    $socialUser = $this->getUserFromSocialite($provider);

    $providerEmail = $socialUser->getEmail();

    if ($this->socialUserHasNoEmail($providerEmail)) {
        throw new EmailNotProvidedException;
    }

    if ($this->socialUserAlreadyLoggedIn()) {
        $this->checkIfAccountSyncedOrSync($socialUser);
    }

    // set authUser from socialUser

    $authUser = $this->setAuthUser($socialUser);

    $this->loginAuthUser($authUser);

    $this->logoutIfUserNotActiveStatus();

    return $this->redirectUser();
}
```

Ok, so now we get into some heavy lifting, but don't worry, we will break it down, method by method.

First we get the verifyProvider method, located in the VerifySocialUsers trait:

```
private function verifyProvider($provider)
{
    if ( ! in_array($provider, $this->approvedProviders)){
        throw new UnauthorizedException();
    }
}
```

All this does is check to see if the \$provider handed in is in the approvedProviders array, which we've set as a property on the ManagesSocialAuth trait:

```
private $approvedProviders = [ 'facebook', 'github'];
```

This is just a little extra security I included, not sure you would really need it, but better safe than sorry. This way no one can spoof a provider.

Next, set the \$socialUser with the getUserFromSocialite method:

```
$socialUser = $this->getUserFromSocialite($provider);
```

The method itself is located in the RoutesSocialUsers trait:

```
private function getUserFromSocialite($provider)
{
    try {
        $socialUser = Socialite::driver($provider)->user();

        return $socialUser;

    } catch (\Exception $e) {

        throw new ConnectionNotAcceptedException;
    }
}
```

We use a try catch block to make sure we have a connection, then use the Socialite::driver(\$provider)->user() methods to return the user object of the social user.

Next method called in the handleProviderCallback in ManagesSocialAuth:

```
$providerEmail = $socialUser->getEmail();
```

The getEmail() method is provided by the Socialite User object \$socialUser.

Next we check to see to make sure the email has a value other than null:

```
if ($this->socialUserHasNoEmail($providerEmail)) {  
    throw new EmailNotProvidedException;  
}
```

The reason for this check is that not all social networks provide email address by default, sometimes that is a user setting, and in cases where the user has decided not to allow their email address to be given, it would come back as null. So we have a way to handle that.

Next, we check to see if the user is already logged in:

```
if ($this->socialUserAlreadyLoggedIn()) {
```

This makes a call to the SocialUserAlreadyLoggedIn method on the VerifiesSocialUsers trait:

```
private function socialUserAlreadyLoggedIn()  
{  
    return Auth::check();  
}
```

You can see we are using Auth::Check to see if the user is logged in. It returns true if logged in, false if not. If the user is logged in and hitting our social route, we assume they want to sync their account or perhaps they are already synced and hit the link again:

```
if ($this->socialUserAlreadyLoggedIn()) {  
  
    $this->checkIfAccountSyncedOrSync($socialUser);  
  
}
```

This makes a call to the checkIfAccountSyncedOrSync() method in the SyncsSocialUsers trait:

```
private function checkIfAccountSyncedOrSync($socialUser)  
{  
  
    //if you are logged in and accountSynced is true, you are  
    //already synced  
  
    if ($this->accountSynced($socialUser)){  
  
        throw new AlreadySyncedException;  
  
    } else {  
  
        // check for email match  
  
        if ( ! $this->authUserEmailMatches($socialUser)) {  
  
            throw new CredentialsDoNotMatchException;  
  
        }  
  
        // if emails match, then sync accounts
```

```
$this->syncUserAccountWithSocialData($socialUser);

alert()->success('Confirmed!', 'You are now synced...');

return $this->redirectUser();

}

}
```

You can see this method is a little more involved. We do a quick check to see if the user is already synced, and if so, throw the appropriate exception:

```
if ($this->accountSynced($socialUser)){

    throw new AlreadySyncedException;

}
```

If the user is not already synced, then we get the else block, which runs us through the methods to sync the account.

First, we check to see if the email matches:

```
if ( ! $this->authUserEmailMatches($socialUser) ) {

    throw new CredentialsDoNotMatchException;

}
```

The authUserEmailMatches method is on the VerifiesSocialUsers Trait, and it's super simple:

```
private function authUserEmailMatches($socialUser)
{
    return $socialUser->email == Auth::user()->email;
}
```

Ok, back to the checkIfAccountSyncedOrSync method on the SyncsSocialUsers trait. Assuming we passed the matching email check, we call the following:

```
$this->syncUserAccountWithSocialData($socialUser);
```

This method is located on the same trait. Let's take a look:

```
private function syncUserAccountWithSocialData($socialUser)
{
    // one last check to see if the social id already exists

    if ($this->socialIdAlreadyExists($socialUser)){
        throw new CredentialsDoNotMatchException;
    }

    // lookup user id and update create provider record

    SocialProvider::create([
        'user_id' => Auth::user()->id,
        'source'   => $this->provider,
        'source_id' => $socialUser->id,
        'avatar'      => $socialUser->avatar
    ]);
}
```

We start with a check to see if the social id already exists:

```
// one last check to see if the social id already exists

if ($this->socialIdAlreadyExists($socialUser)){

    throw new CredentialsDoNotMatchException;

}
```

This calls the socialIdAlreadyExists method in the VerifiesSocialUsers trait:

```
private function socialIdAlreadyExists($socialUser)
{
    return SocialProvider::where('source_id', $socialUser->id)->exists();
}
```

Here we are using eloquent's exists method to return true or false, it does not return the actual record.

So if we pass that check, back in the syncUserAccountWithSocialData method, we use eloquent to create the record that associates the accounts:

```
SocialProvider::create([
    'user_id' => Auth::user()->id,
    'source'   => $this->provider,
    'source_id' => $socialUser->id,
    'avatar'   => $socialUser->avatar
]);
```

If you will recall, the syncUserAccountWithSocialData method was called from within the checkIfAccountSyncedOrSync method, so let's go back to that to finish up this part:

```
alert()->success('Confirmed!', 'You are now synced...');

return $this->redirectUser();
```

We throw an alert and then redirectUser via that method located in the RoutesSocialUsers trait:

```
private function redirectUser()
{
    if (Auth::user()->isAdmin()){

        return redirect()->route('admin');

    }

    return redirect()->route('home');
}
```

We're just checking to see if the user is admin or not and redirecting appropriately.

So now we go back to the handleProviderCallback method in the ManagesSocialAuth trait, and pick up where we left off. I'm including the previous block for reference:

```
if ($this->socialUserAlreadyLoggedIn()) {  
  
    $this->checkIfAccountSyncedOrSync($socialUser);  
  
}
```

This is what we just completed doing. Let's continue:

```
// set authUser from socialUser  
  
$authUser = $this->setAuthUser($socialUser);  
  
$this->loginAuthUser($authUser);  
  
$this->logoutIfUserNotActiveStatus();  
  
return $this->redirectUser();
```

So, assuming we passed the point where we know the user is not logged in, we run:

```
$authUser = $this->setAuthUser($socialUser);
```

The setAuthUser method is in the SetsAuthUser trait:

```
private function setAuthUser($socialUser)
{
    $authUser = $this->findOrCreateUser($socialUser);

    return $authUser;

}
```

This calls the findOrCreateUser method from the FindsOrCreatesUsers Trait:

```
private function findOrCreateUser($socialUser)
{
    // is email already in table?

    if ( $authUser = $this->userWhereEmailMatches($socialUser)) {

        // scenario where email is already in table
        // is the provider source correct and does the social id match?
        // if there is a match, return $authUser, if not throw exception

        if ( ! $this->matchesIds($socialUser, $authUser)) {

            // exception instructs user to login first, then sync
            // covers scenario where there are matching emails,
            // but no matching id
            // or where there is existing social id and it
            // doesn't match

            throw new EmailAlreadyInSystemException;

        }

        // if email and id matches, return the $authUser

    }

    return $authUser;
}
```

```
}

// scenario where no matching email,
// but social id already exists

if ($this->socialIdAlreadyExists($socialUser)){

    throw new CredentialsDoNotMatchException;

}

// if no user matching social exists, we create one

$authUser = $this->makeNewUser($socialUser);

return $authUser;

}
```

I heavily commented this method, it does a lot. Let's step through it. We start by checking if the email of the social user is already in the users table:

```
if ( $authUser = $this->userWhereEmailMatches($socialUser)){

    // scenario where email is already in table
    // is the provider source correct and
    // does the social id match?
    // if there is a match, return $authUser,
    // if not throw exception

    if ( ! $this->matchesIds($socialUser, $authUser)) {

        // exception instructs user to login first, then sync
        // covers scenario where there are matching emails,
        // but no matching id
        // or where there is existing social id and it doesn't match

        throw new EmailAlreadyInSystemException;
    }
}
```

```
}
```

If the email already exists, we check to see if matches the ids of the social user and a user in our users table via the matchesIds method from the VerifiesSocialUsers Trait:

```
private function matchesIds($socialUser, User $authUser)
{
    return $authUser->socialProviders()
        ->where('source', $this->provider)
        ->where('source_id', $socialUser->id)
        ->first();
}
```

What's interesting here is that we are using a relationship to do the query. You can see from the type hint in the method signature that \$authUser is an instance of User. And if you remember, we created the has many socialProviders method on the User model, which means we can very simply access a user and their related social provider record.

We match the source to \$this->provider, which is a property set on the ManagesSocialAuth trait, and we also match the ids of the users.

So if the whole statement finds a matching user, where the emails and ids match, then we return the user:

```
return $authUser;
```

So then we login and execute the rest of the handleProviderCallback method:

```
$this->loginAuthUser($authUser);

$this->logoutIfUserNotActiveStatus();

return $this->redirectUser();
```

We will cover those in detail, but before we do, let's cover the scenario where it's a new user who is not already in the system. For that we go back to our `findOrCreateUser` method in our `FindsOrCreatesUsers` trait and pick up where left off previously:

```
// scenario where no matching email,
// but social id already exists

if ($this->socialIdAlreadyExists($socialUser)){

    throw new CredentialsDoNotMatchException;

}

// if no user matching social exists, we create one

$authUser = $this->makeNewUser($socialUser);

return $authUser;
```

First we check the table to see if the id already exists, in which case we will not allow the registration to complete. We do this check via the `socialIdAlreadyExists` method in the `VerifiesSocialUsers` trait:

```
private function socialIdAlreadyExists($socialUser)
{
    return SocialProvider::where('source_id', '=', $socialUser->id)->exists();
}
```

If I haven't mentioned it before , the exists() eloquent method does not return the record, it returns true or false.

So if the socialIdAlreadyExists method returns false, we can make the new user:

```
// if no user matching social exists, we create one
$authUser = $this->makeNewUser($socialUser);
```

This calls the makeNewUser method on the same trait:

```
private function makeNewUser($socialUser)
{
    //create user if not already exists and email does not
    //already exist.

    $password = $this->makePassword();

    DB::beginTransaction();

    try{
        $authUser = User::create([
            'name' => $socialUser->name,
            'email' => $socialUser->email,
            'password' => $password,
            'status_id' => 10,
        ]);
    }
```

```
SocialProvider::create([
    'user_id' => $authUser->id,
    'source'   => $this->provider,
    'source_id' => $socialUser->id,
    'avatar'   => $socialUser->avatar,
]);

DB::commit();

} catch (\Exception $e) {
    DB::rollback();

    throw new TransactionFailedException();
}

return $authUser;
}
```

First we create a password for the user:

```
$password = $this->makePassword();
```

Database Transactions

Next, we begin a transaction:

```
DB::beginTransaction();
```

In this case, we have records that have to be created for two tables, the users table and the social_providers table, and we don't want to insert any records unless all inserts are successful.

The way we insure that is to wrap the two create calls in a transaction and a try/catch block:

```

try{
    $authUser = User::create([
        'name' => $socialUser->name,
        'email' => $socialUser->email,
        'password' => $password,
        'status_id' => 10,
    ]);

    SocialProvider::create([
        'user_id' => $authUser->id,
        'source' => $this->provider,
        'source_id' => $socialUser->id,
        'avatar' => $socialUser->avatar,
    ]);

    DB::commit();
}

catch (\Exception $e) {
    DB::rollback();
    throw new TransactionFailedException();
}

```

If the try is successful, we run DB::commit. If it fails, we run DB::rollback and throw a TransactionFailedException.

So that's a quick run through a transaction, you can see how easy Laravel makes this for us. You can read more about transactions in the [docs](#).

And finally, if we have run a successful transaction, we return the \$authUser:

```
return $authUser;
```

Ok, so now we can go back to the final part of our handleProviderCallback in our ManagesSocialAuth trait:

```
$this->loginAuthUser($authUser);

$this->logoutIfUserNotActiveStatus();

return $this->redirectUser();
```

From here, we loginAuthUser from that method on the RoutesSocialUsers trait:

```
private function loginAuthUser($authUser)
{
    Auth::login($authUser, true);
}
```

Next we check the status and if they are not active, immediately log them out:

```
$this->logoutIfUserNotActiveStatus();
```

And finally, if we pass that check, we redirectUser() to the appropriate page.

I did my best to make this implementation of Socialite as intuitive as possible. I've divided up a lot of methods into various traits, which will make it easier for me to maintain, and I've also layered this approach to auth directly on top of what Laravel gives us out of the box to maximize compatibility moving forward.

Now if you want to add a second social network, let's say Github, you are most of the way there. We need to add just a little bit more to account for differences in what the social provider is going to return.

We already have a check for email, and since email is a unique identifier, we throw an exception if they don't have one.

When I tested my implementation on Github, I found that I had null for the name. I guess I never filled it out. But I did have a nickname that was being returned.

This underscores the fact that different providers may return fields with names that don't exactly fit our schema. So what can we do?

Let's start by adding a property to our ManagesSocialAuth.php trait:

```
private $userName;
```

Next we'll call a method to set this property in the handleProviderCallback method. Put it below the first if statement like so:

```
if ($this->socialUserHasNoEmail($providerEmail)) {  
  
    throw new EmailNotProvidedException;  
  
}  
  
$this->setSocialUserName($socialUser);
```

Here is the full gist of the ManagesSocialAuth trait for reference:

ManagesSocialAuth

Next we need to create the setSocialUserName method in the SetsAuthUser trait:

Gist:

setSocialUserName

```
private function setSocialUserName($socialUser)  
{  
  
    switch ($this->provider){  
  
        case 'github' :  
  
            is_null($socialUser->name) ?  
  
                $this->userName = $socialUser->nickname :  
  
                $this->userName = $socialUser->name;  
  
        break;  
    }
```

```

case 'facebook' :

    $this->userName = $socialUser->name;

    break;

default :

    is_null($socialUser->name) ?

        $this->userName = $this->createUserName() :

        $this->userName = $socialUser->name;

    break;

}

```

You can see we are using a switch statement and a ternary to set the \$userName. In the case of Github, if there is no name value set, then we use the nickname value.

I also thought it was prudent to set up a createUserName method to default to if we want to use that option for a provider, so add this method to the setsAuthUser trait:

```

private function createUserName()
{
    return str_random(15);
}

```

We just use Laravel's str_random method, which is a global helper, to make the string for us. You can check out more helpers from Laravel in the [docs](#).

Here is the gist for the entire SetsAuthUser trait:

SetsAuthUser

Ok, one last change inside the makeNewUser method in the FindsOrCreatesUsers trait:

```
$authUser = User::create([  
  
    'name' => $this->userName,  
    'email' => $socialUser->email,  
    'password' => $password,  
    'status_id' => 10,  
  
]);
```

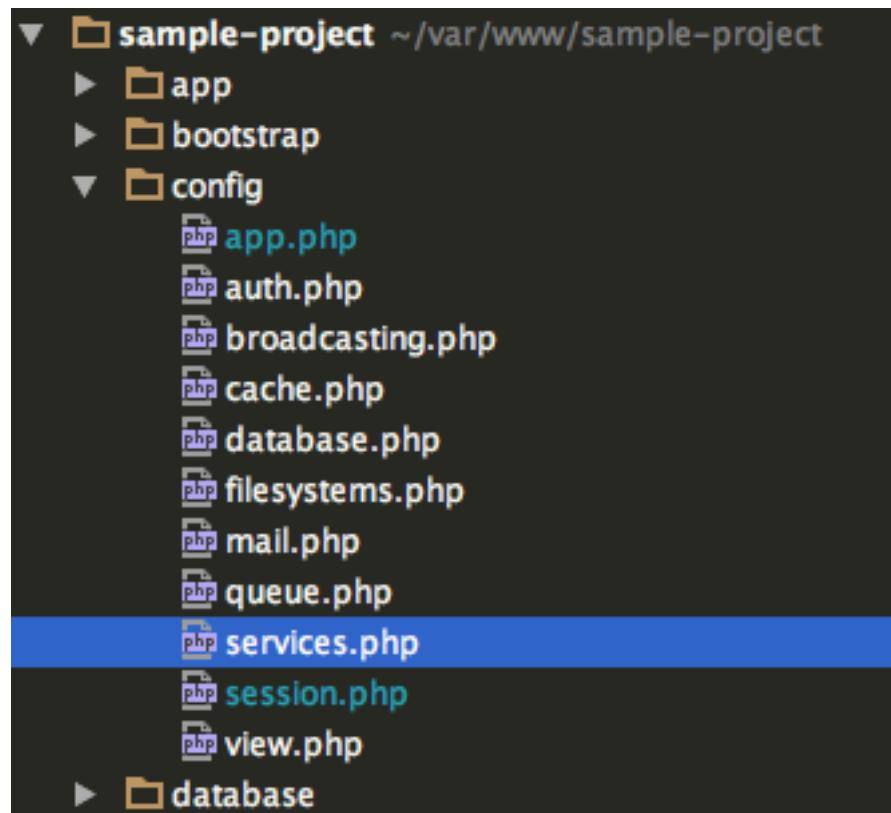
You can see that we changed the ‘name’ field to use `$this->userName` instead of `$socialUser->name`. If we didn’t do that and you had a null value for name, the transaction would fail.

So at this point, we’ve given ourselves the ability to customize which value goes into the name field, depending on the provider. When you set up a new provider, go ahead and `dd($socialUser)` in the `handleProviderCallback` method to see what it is returning and you can decide how to format from there.

Since we went through all of this, let’s go ahead and add Github as a provider. I know I said we were only going to do Facebook, but this will show us how useful all the work we have already done is, so I think it’s worth it to step through it.

If you don’t already have a Github account, sign up for one, it’s free.

We’ll start by adding the following to `config/services.php` located here:



```
'github' => [  
  
    'client_id' => env('GITHUB_ID'),  
    'client_secret' => env('GITHUB_SECRET'),  
    'redirect' => env('GITHUB_URL'),  
  
,
```

Then in your .env file:

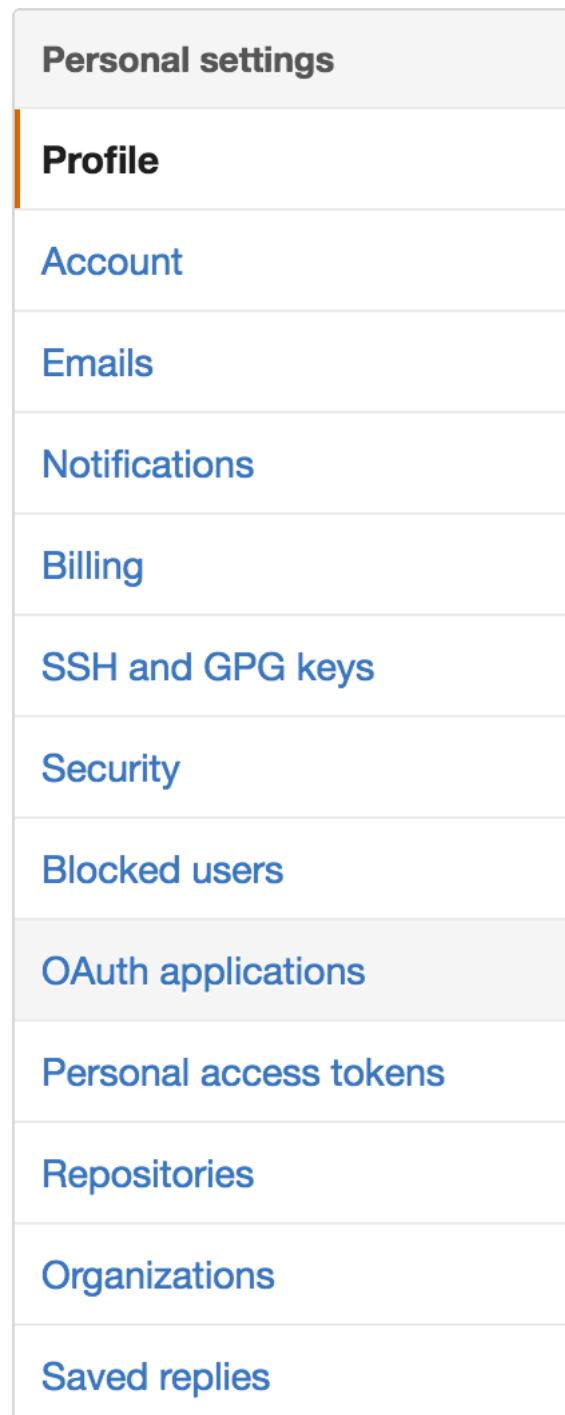
```
GITHUB_ID=your-github-id  
GITHUB_SECRET=your-github-secret  
GITHUB_URL=http://sample-project.com/auth/github/callback
```

I'm going to give you the instructions for application setup from Github, but again I have to warn you it's very likely that the screenshots can be out of date and not accurate. In that case, just do your best to follow Github's instructions, at least what I have provided here will give you some idea of how to do it.

Assuming you have a Github account, you can get your app set up at the following url:

<https://github.com/settings/profile>

Then click on OAuth Applications in the left nav:



You will get a page with a Register a new application button.

Click on the button and that will lead you to the following:

The screenshot shows the GitHub 'Personal settings' sidebar on the left and the main 'Register a new OAuth application' form on the right. The sidebar includes links for Profile, Account, Emails, Notifications, Billing, SSH and GPG keys, Security, Blocked users, Repositories, Organizations, Saved replies, Authorized applications, and Installed integrations. The main form has fields for Application name (set to 'sample-project'), Description (set to 'Something users will recognize and trust'), Homepage URL (set to 'http://sample-project.com'), Application description (set to 'Sample Project'), and Authorization callback URL (set to 'http://sample-project.com/auth/github/callback'). A green 'Register application' button is at the bottom.

Obviously, you will be using your own values for the above.

Once you submit that page, you land on a page with your app id and secret. Copy those values into your .env file.

Now when you try the following url:

```
sample-project.com/auth/github
```

You should be able to sync your account or register a new user if you don't already have an account.

Summary

Hopefully this has all gone smoothly for you. Socialite is easy to work with, relatively easy to work with, if you've got your environment working correctly and if you've set up your 3rd party app correctly.

We wrote in numerous custom exceptions to help us diagnose problems and also to help the user by returning a nicely formatted exception that gives them instructions. For example, if they previously had an account, they need to login first, before they can sync their accounts and the view for the exception tells them that.

The AuthController got a bit more complex because it's doing the job of both the LoginController and the RegisterController as well as taking in the social auth methods from the ManagesSocialAuth trait. We moved the many of the trait methods to nested traits, which cut down on the code bloat.

Chapter 9: Profile, Settings and Admin Dash

Our example project is coming along nicely, but we still need to build the basics. Most sites will have some sort of profile for users. Also, we typically see some way for the user to update their settings with things like their username, email, password, etc.

Our admin dash should also give us control over the users and their profiles.

Profile

Ok, so we'll build our Profile model first, and do it in the following order:

1. Make Model and migration.
2. Fill out our migration and migrate.
3. Create route.
4. Create controller.
5. Create views.

We will also be drawing from our Widgets example when we can. While it's not a perfect match, there are some elements we can pull from widget template, so that will be really useful.

So let's start by making a model and migration.

```
php artisan make:model Profile -m
```

Let's move on to the migration and change it to the following:

Gist:

[profile migration](#)

From book:

```
<?php

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

class CreateProfilesTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */

    public function up()
    {
        Schema::create('profiles', function (Blueprint $table) {

            $table->increments('id');
            $table->integer('user_id')->unsigned();
            $table->string('first_name', 60);
            $table->string('last_name', 60);
            $table->date('birthdate');
            $table->boolean('gender');
            $table->timestamps();

        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */

    public function down()
    {
        Schema::dropIfExists('profiles');

    }
}
```

Ok, so you can see what we are planning by the column names. A user profile will have a first name, a last name, birthdate, and gender. We will also have the typical auto-increment id and timestamps as well as a user_id column to tie it to the specific user it relates to.

Let's go ahead and migrate this up:

```
php artisan migrate
```

And now you should have a profiles table that looks like this:

	#	Name	Type	Collation	Attributes	Null	Default	Extra	Action
<input type="checkbox"/>	1	id	int(10)		UNSIGNED	No	<i>None</i>	AUTO_INCREMENT	Change Drop
<input type="checkbox"/>	2	user_id	int(10)		UNSIGNED	No	<i>None</i>		Change Drop
<input type="checkbox"/>	3	first_name	varchar(60)	utf8_unicode_ci		No	<i>None</i>		Change Drop
<input type="checkbox"/>	4	last_name	varchar(60)	utf8_unicode_ci		No	<i>None</i>		Change Drop
<input type="checkbox"/>	5	birthdate	date			No	<i>None</i>		Change Drop
<input type="checkbox"/>	6	gender	tinyint(1)			No	<i>None</i>		Change Drop
<input type="checkbox"/>	7	created_at	timestamp			Yes	<i>NULL</i>		Change Drop
<input type="checkbox"/>	8	updated_at	timestamp			Yes	<i>NULL</i>		Change Drop

↑ Check All With selected: Browse Change Drop Primary Unique Index

Print view Propose table structure Move columns Improve table structure

Great. Now we need to make sure we can mass-assign on the model. We're also going to add the relationship to user and we're going to define 2 Accessors. I will explain everything in detail, but first here is the file.

Gist

[Profile.php](#)

From book:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Profile extends Model
{
    protected $fillable =['user_id',
                         'first_name',
                         'last_name',
                         'birthdate',
                         'gender'];

    protected $dates = [ 'birthdate'];

    public function showGender($gender)
    {
        return $gender == 1 ? 'Male' : 'Female';
    }

    public function fullName()
    {
        return $this->first_name . ' ' . $this->last_name;
    }

    public function user()
    {
        return $this->belongsTo('App\User');
    }
}
```

You'll note we are not extending SuperModel because we do not need the created_at attribute formatted for us.

So let's step through the model. We have our \$fillable property for our mass-assignable attributes:

```
protected $fillable =[ 'user_id',
                      'first_name',
                      'last_name',
                      'birthdate',
                      'gender'];
```

We have a property named dates:

```
protected $dates = [ 'birthdate'];
```

In that array, we have the birthday attribute. This tells eloquent to return this attribute as a carbon instance, so we can do things like:

```
$profile->birthdate->format('m-d-Y')
```

We can easily get that format from an accessor, like we did with the SuperModel class, but this is another way to give yourself flexibility and allows you to change the format if you wish.

Then we get our first method.

```
public function showGender($gender)
{
    return $gender == 1 ? 'Male' : 'Female';
}
```

We are returning simple ternary statement to give us the string value of gender.

Note that we don't have any hard logic somewhere, let's say by creating a constant, defining Male and Female. You could go that route if you wish. I felt that this was simpler and that the constants weren't necessary.

Next we have a method to concatenate first_name and last_name to give us the full name of the user, which will make it easier for us to display in the views:

```
public function fullName()
{
    return $this->first_name . ' ' . $this->last_name;

}
```

Then to wrap up the model, we have the relationship to User:

```
public function user()
{
    return $this->belongsTo('App\User');

}
```

One profile belongs to one user. It has baked in logic that will match user_id on Profile to User id. It really is very simple when you start to get it.

In workflow, it's a good idea to do the matching relationship, so while we are thinking about it, add this to your User model, User.php:

```
public function profile()
{
    return $this->hasOne('App\Profile');
}
```

Ok, let's make a route for profile in our routes/web.php file. We'll use a simple route resource:

```
Route::resource('profile', 'ProfileController');
```

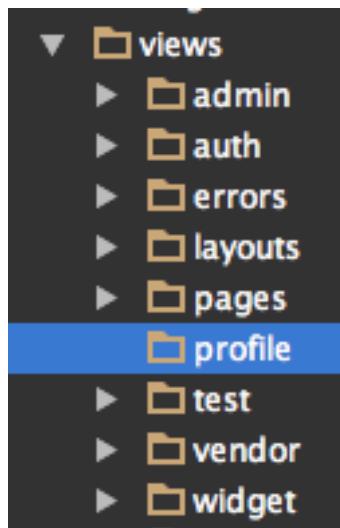
We've already covered a route resource, it will create all the routes we need for the RESTful pattern.

Next, let's create a matching controller:

```
php artisan make:controller ProfileController --resource
```

Since this is a boilerplate stubbed out controller, I won't show the code until we start making the methods.

Now let's create a folder for our views. In the views folder, create a folder named profile.



Before we work on the view, we're going to pull in a package that we will need for dealing with the date input on the form. You can get the full instructions and docs for the package here:

[Laravel Collective](#)

We are going to install the html package.

From your command line run:

```
composer require laravelcollective/html
```

Then in config/app.php add to your providers array:

```
Collective\Html\HtmlServiceProvider::class,
```

Then in your aliases array, add the following:

```
'Form' => Collective\Html\FormFacade::class,  
'Html' => Collective\Html\HtmlFacade::class,
```

In previous books and in my applications, I used the form helper a lot, but I don't use it much any more. The reason is like to keep my html clean, which makes implementing javascript a lot easier. But in the case of working with date input, I still prefer the helper, since I had real trouble manipulating the date input without it.

Ok, so now we're going to start with the create view. With the exception of the actual form inputs, most of this is exactly like the widget create view, so you can reference that when building the view. For now, we'll jump ahead and I'll give you the file:

Gist:

[create.blade.php](#)

From book:

```
@extends('layouts.master')  
  
@section('title')  
  
<title>Create a Profile</title>  
  
@endsection  
  
@section('content')  
  
<ol class='breadcrumb'>  
    <li><a href='/'>Home</a></li>  
    <li class='active'>Create Profile</li>  
</ol>  
  
<h2>Create Your Profile</h2>
```

```
<hr/>

<form class="form"
      role="form"
      method="POST"
      action="{{ url('/profile') }}>

    {{ csrf_field() }}

<!-- first_name Form Input -->

<div class="

    form-group{{ $errors->has('first_name') ? ' has-error' : '' }}>

    <label class="control-label">First Name</label>

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

    @if ($errors->has('first_name'))

        <span class="help-block">

            <strong>{{ $errors->first('first_name') }}</strong>

        </span>

    @endif

</div>

<!-- last_name Form Input -->

<div class="

    form-group{{ $errors->has('last_name') ? ' has-error' : '' }}>

    </>
```

```
<label class="control-label">Last Name</label>

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

@if ($errors->has('last_name'))

    <span class="help-block">

        <strong>{{ $errors->first('last_name') }}</strong>

    </span>

@endif

</div>

<!-- birthdate Form Input -->

<div class="

    form-group{{ $errors->has('birthdate') ? ' has-error' : '' }}>

    <label class="control-label">Birthdate</label>

    <div>

        {{ Form::date('birthdate') }}

    </div>

    @if ($errors->has('birthdate'))

        <span class="help-block">

            <strong>{{ $errors->first('birthdate') }}</strong>

        </span>

    @endif
```

```
</div>

<!-- Gender Form Input -->

<div class="

    form-group{{ $errors->has('gender') ? ' has-error' : '' }}>

    <label class="control-label">Gender</label>

    <select class="form-control"
        id="gender"
        name="gender">

        <option value="{{ old('gender') }}>

            {{ ! is_null(old('gender')) ?
                (old('gender') == 1 ? 'Male' : 'Female')
                : 'Please Choose One' }}</option>

        <option value="1">Male</option>
        <option value="0">Female</option>
    </select>

    @if ($errors->has('gender'))
        <span class="help-block">
            <strong>{{ $errors->first('gender') }}</strong>
        </span>
    @endif
</div>
```

```
<div class="form-group">

    <button type="submit" class="btn btn-primary btn-lg">
        Create
    </button>

</div>

</form>

@endsection
```

So we'll skip over the parts we already know. We have a date input, using the form helper we just installed:

```
<div class="

    form-group{{ $errors->has('birthdate') ? ' has-error' : '' }}>

    <label class="control-label">Birthdate</label>

    <div>

        {{ Form::date('birthdate') }}

    </div>

@if ($errors->has('birthdate'))

    <span class="help-block">

        <strong>{{ $errors->first('birthdate') }}</strong>

    </span>

@endif
```

```
</div>
```

The form helper is nice and concise:

```
{{ Form::date('birthdate') }}
```

You just give the name of the attribute, in this case birthdate.

Next, we have a dropdown list for Gender:

```
<div class="form-group{{ $errors->has('gender') ? ' has-error' : '' }}>

    <label class="control-label">Gender</label>

    <select class="form-control"
        id="gender"
        name="gender">

        <option value="{{old('gender')}}">
            {{ ! is_null(old('gender')) ? (old('gender') == 1 ? 'Male' : 'Female') : 'Please Choose One'}}</option>

        <option value="1">Male</option>

        <option value="0">Female</option>

    </select>

    @if ($errors->has('gender'))
```

```
<span class="help-block">  
    <strong>{{ $errors->first('gender') }}</strong>  
</span>  
  
@endif  
</div>
```

We use a nested ternary to check to see if we have an old value, and if so, display that value, if not display “Please Choose One.”

We’re only hardcoding the other select values because the list is short and it’s not going to change.

If you had a longer list of values for the select that you wanted to grab from the DB, eloquent has a really cool way to do that. It’s worth mentioning, even though we don’t need it here.

Do not use this code, it’s only an example.

Let’s say we were doing a select for categories. In the controller, we would do something like the following:

```
$categories = Category::pluck('name', 'id');
```

The pluck method returns an array with only the named columns, category name and id. The other columns on the table would be ignored.

Then pass \$categories to the view like so:

```
return view('yourview.create', compact('categories'));
```

Then inside the select, to list the options, it would look something like this:

```
@foreach($categories as $category)  
    <option value="{{ $category->id }}>{{ $category->name }}</option>  
@endforeach
```

You can see how easy that is to work with.

Let's move on to our ProfileController and change the create method to the following:

```
public function create()  
{  
    return view('profile.create');  
}
```

Ok, so now if you point your browser to sample-project.com/profile/create, you should get the following:

Sample Project

Home About Content Bill Keck

Home / Create Profile

Create Your Profile

First Name

Last Name

Birthdate

 mm/dd/yyyy

Gender

Please Choose One

Create

© 2015 - 2017 Sample Project All rights Reserved.

If you object to the input boxes spanning all the way across, feel free to adjust the css as you see fit. I'm doing minimal UI work intentionally.

Anyway, you can see it's nice and clean.

Moving on, I would normally just grab the store method from the WidgetController and use that to get us started. But our ProfileController has significant differences, so at this point, I need to give you the complete controller. It will be confusing if you don't have the complete file to refer to when we're working on this, especially with the use statements. So here we go:

Gist:

[ProfileController](#)

From book:

```
<?php

namespace App\Http\Controllers;

use App\Http\Auth\Traits\OwnsRecord;
use Illuminate\Http\Request;
use App\Profile;
use App\User;
use Redirect;
use Illuminate\Support\Facades\Auth;
use DB;
use App\Exceptions\UnauthorizedException;

class ProfileController extends Controller
{
    use OwnsRecord;

    public function __construct()
    {
        $this->middleware('auth');
        $this->middleware('admin',['only'=> 'index']);
    }

    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        $profiles = Profile::paginate(10);

        return view('profile.index', compact('profiles'));
    }

    public function determineProfileRoute()
    {
        $profileExists = $this->profileExists();
    }
}
```

```
if ($profileExists){

    return Redirect::route('show-profile');

}

return view('profile.create');

}

public function showProfileToUser()
{

$profile = Profile::where('user_id', Auth::id())->first();

if( ! $profile){

    return Redirect::route('profile.create');

}

$user = User::where('id', $profile->user_id)->first();

if ($this->userNotOwnerOf($profile)) {

    throw new UnauthorizedException;
}

return view('profile.show', compact('profile', 'user'));

}

/**
 * Show the form for creating a new resource.
 *
 * @return \Illuminate\Http\Response
 */

public function create()
{
    $profileExists = $this->profileExists();
```

```
if ($profileExists){

    return Redirect::route('show-profile');

}

return view('profile.create');
}

/**
 * Store a newly created resource in storage.
 *
 * @param \Illuminate\Http\Request $request
 * @return \Illuminate\Http\Response
 */

public function store(Request $request)
{

    $this->validate($request, [
        'first_name' => 'required|alpha_num|max:20',
        'last_name' => 'required|alpha_num|max:20',
        'gender' => 'boolean|required',
        'birthdate' => 'date|required',
    ]);

    $profileExists = $this->profileExists();

    if ($profileExists){

        return Redirect::route('show-profile');

    }

    $profile = Profile::create(['first_name' => $request->first_name,
        'last_name' => $request->last_name,
        'gender' => $request->gender,
        'birthdate' => $request->birthdate,
        'user_id' => Auth::user()->id]);
    $profile->save();
}
```

```
$user = User::where('id', '=', $profile->user_id)->first();

alert()->success('Congrats!', 'You made your profile');

return view('profile.show', compact('profile', 'user'));

}

/** 
 * Display the specified resource.
 *
 * @param int $id
 * @return \Illuminate\Http\Response
 */
public function show($id)
{
    $profile = Profile::findOrFail($id);

    $user = User::where('id', $profile->user_id)->first();

    if( ! $this->adminOrCurrentUserOwns($profile)){
        throw new UnauthorizedException;
    }

    return view('profile.show', compact('profile', 'user'));
}

/** 
 * Update the specified resource in storage.
 *
 * @param \Illuminate\Http\Request $request
 * @param int $id
 * @return \Illuminate\Http\Response
 */
public function edit($id)
{
    $profile = Profile::findOrFail($id);

    if ( ! $this->adminOrCurrentUserOwns($profile)){
        throw new UnauthorizedException;
    }

    $profile->fill($request->all());
    $profile->save();
}
```

```
        throw new UnauthorizedException;

    }

    return view('profile.edit', compact('profile'));
}

/**
 * Update the specified resource in storage.
 *
 * @param \Illuminate\Http\Request $request
 * @param int $id
 * @return \Illuminate\Http\Response
 */

public function update(Request $request, $id)
{
    $this->validate($request, [
        'first_name' => 'required|alpha_num|max:20',
        'last_name' => 'required|alpha_num|max:20',
        'gender' => 'boolean|required',
        'birthdate' => 'date|required'
    ]);

    $profile = Profile::findOrFail($id);

    if ($this->userNotOwnerOf($profile)) {

        throw new UnauthorizedException;
    }
    $profile->update(['first_name' => $request->first_name,
                      'last_name' => $request->last_name,
                      'gender' => $request->gender,
                      'birthdate' => $request->birthdate]);

    alert()->success('Congrats!', 'You updated your profile');

    return Redirect::route('profile.show', [
        'profile' => $profile
    ]);
}
```

```
}

/**
 * Remove the specified resource from storage.
 *
 * @param int $id
 * @return \Illuminate\Http\Response
 */

public function destroy($id)
{
    $profile = Profile::findOrFail($id);

    if ($this->userNotOwnerOf($profile)){

        throw new UnauthorizedException;
    }

    Profile::destroy($id);

    if (Auth::user()->isAdmin()){

        alert()->overlay('Attention!',
            'You deleted a profile', 'error');

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

    alert()->overlay(
        'Attention!', 'You deleted a profile', 'error'
    );

    return Redirect::route('home');
}

/**
 * @return mixed
 */

```

```
private function profileExists()
{
    $profileExists = DB::table('profiles')
        ->where('user_id', Auth::id())
        ->exists();

    return $profileExists;
}

}
```

Now that we have that as a reference, it will be easier for us to talk about the methods in context, now that we have our use statements and our trait pulled in:

```
use App\Http\Auth\Traits\OwnsRecord;
use Illuminate\Http\Request;
use App\Profile;
use App\User;
use Redirect;
use Illuminate\Support\Facades\Auth;
use DB;
use App\Exceptions\UnauthorizedException;

class ProfileController extends Controller
{
    use OwnsRecord;
```

So let's take a minute to discuss what will be different about our ProfileController. For the user experience, we want a single link that they click for profile. The controller will then determine if the user already has a profile, if so, show it, otherwise, take the user to the create page. To do this, we need a couple of methods we haven't seen before. The first one is determineProfileRoute():

```
public function determineProfileRoute()
{
    $profileExists = $this->profileExists();

    if ($profileExists){
        return Redirect::route('show-profile');

    }

    return view('profile.create');
}
```

This is our check to see if they already have a profile. We use the profileExists method to check it:

```
private function profileExists()
{
    $profileExists = DB::table('profiles')
        ->where('user_id', Auth::id())
        ->exists();

    return $profileExists;
}
```

So we query the profiles table to see if they already have one, using Auth::id();

Note the use of Laravel's query builder's exists() method.

Back in our determineProfileRoute method, you can see we use this test to decide where to route the user:

```
public function determineProfileRoute()
{
    $profileExists = $this->profileExists();

    if ($profileExists){
        return Redirect::route('show-profile');

    }

    return view('profile.create');
}
```

This is also a way to get the user's profile, if they have one, without passing in the id of the record.

If the profile does not exist, we return the create view:

```
return view('profile.create');
```

We will need to add the following routes for this to work.

determine-profile-route and show-profile Routes

These will need to go above the route resource, which I'm showing for reference:

```
Route::get('show-profile',
'ProfileController@showProfileToUser')->name('show-profile');

Route::get('determine-profile-route',
'ProfileController@determineProfileRoute')
->name('determine-profile-route');

Route::resource('profile', 'ProfileController');
```

When we use the show-profile route, we get the showProfileToUser method:

```
public function showProfileToUser()
{
    $profile = Profile::where('user_id', Auth::id())->first();

    if( ! $profile){
        return Redirect::route('profile.create');
    }

    $user = User::where('id', $profile->user_id)->first();

    if ($this->userNotOwnerOf($profile)){
        throw new UnauthorizedException;
    }

    return view('profile.show', compact('profile', 'user'));
}
```

So the first thing it does is figure out the correct profile id to show to the user:

```
$profile = Profile::where('user_id', Auth::id())->first();
```

By doing it this way, we get an added layer of security. Since we are not handing in the profile id, there is no way to spoof it. If for whatever reason the user got access to this route and didn't have a profile, we redirect to the create route:

```
if( ! $profile){

    return Redirect::route('profile.create');

}
```

To finish the method, we grab the user, so we can pass it along to the view, then do a check to make sure the person viewing the profile is the owner or admin, then return the show view, with the user and profile instances compacted.

```
$user = User::where('id', $profile->user_id)->first();

if ($this->userNotOwnerOf($profile)){

    throw new UnauthorizedException;
}

return view('profile.show', compact('profile', 'user'));
```

So why have a show method if we have that? The show method requires an id, and as part of the RESTful pattern, we left it in. This gives us a second path to the individual profile records, which can be accessed by admin or the user that owns the record.

We'll be seeing that in action when we build our datagrid for profiles. Let's look at the show method now:

```
public function show($id)
{
    $profile = Profile::findOrFail($id);

    $user = User::where('id', $profile->user_id)->first();

    if( ! $this->adminOrCurrentUserOwns($profile)){
        throw new UnauthorizedException;
    }

    return view('profile.show', compact('profile', 'user'));
}
```

So the only thing we have here that we wouldn't see on a more open record is:

```
if( ! $this->adminOrCurrentUserOwns($profile)){
    throw new UnauthorizedException;
}
```

Hopefully that is clear.

Note that we added the following block to the determineProfileRoute, store and create methods:

```
$profileExists = $this->profileExists();  
  
if ($profileExists){  
  
    return Redirect::route('show-profile');  
  
}
```

That stops us from circumventing the navigation to create more than one profile.

You also see the following block on numerous methods:

```
if ($this->userNotOwnerOf($profile)){  
  
    throw new UnauthorizedException;  
  
}
```

Our super simple syntax explains itself.

We are also using the method that allows admin to execute the action, even if they are not the owner:

```
if( ! $this->adminOrCurrentUserOwns($profile)){  
  
    throw new UnauthorizedException;  
}
```

Again, super easy to follow.

I'm going to skip over the edit and update methods, you can see from the code we have already covered that ground. Instead, let's talk about the destroy method:

```
public function destroy($id)
{
    $profile = Profile::findOrFail($id);

    if ($this->userNotOwnerOf($profile)){
        throw new UnauthorizedException;
    }

    Profile::destroy($id);

    if (Auth::user()->isAdmin()){

        alert()->overlay(
            'Attention!', 'You deleted a profile', 'error'
        );

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

    alert()->overlay(
        'Attention!', 'You deleted a profile', 'error'
    );

    return Redirect::route('home');
}
```

Here I've decided to limit deleting the profile to the owner of the profile. You might want to give admin that privilege as well. If so, use:

```
if ( ! $this->adminOrCurrentUserOwns($profile) ) {  
  
    throw new UnauthorizedException;  
  
}
```

The only other thing to note on the method is that we redirect the user to two different locations, depending on whether or not they are admin.

```
if (Auth::user()->isAdmin()){  
  
    alert()->overlay(  
  
        'Attention!', 'You deleted a profile', 'error'  
  
    );  
  
    return Redirect::route('profile.index');  
}  
  
alert()->overlay(  
  
    'Attention!', 'You deleted a profile', 'error'  
  
);  
  
return Redirect::route('home');
```

Add Profile to Nav

Before we move on with our views, let's take a moment to add our profile link to our nav. Place the following above the Facebook Sync in nav.blade.php:

```
<li><a href="/determine-profile-route">Profile</a></li>
```

I will give you the Gist of the entire file for reference:

[nav.blade.php](#)

Profile Views

Ok, we're ready to move on to our profile views. We already have the create view. Let's do the show view next.

Show View - Profile

For the sake of those working offline, I will show the code in the book, but we will only discuss things we haven't gone over before. In your views/profile folder, make a file named show.blade.php with the following contents:

Gist:

[show.blade.php](#)

From book:

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

@section('title')
    <title>Profile</title>

@endsection

@section('content')
    @if(Auth::user()->isAdmin())
        <ol class='breadcrumb'>
            <li><a href='/'>Home</a></li>
            <li><a href='/profile'>Profiles</a></li>
            <li><a href='/profile/create'>Create</a></li>
        </ol>
    @endif

```

```
@else

<ol class='breadcrumb'>
  <li><a href='/'>Home</a></li>
  <li><a href='/profile/create'>Create</a></li>
</ol>

@endif

<h1>{{ $profile->fullName() }}</h1>

<hr/>

<div class="panel panel-default">

  <!-- Table -->

  <table class="table table-striped">

    <thead>
      <tr>

        <th>Id</th>
        <th>Name</th>
        <th>Gender</th>
        <th>Birthdate</th>

        @if(Auth::user()->adminOrCurrentUserOwns($profile))

        <th>Edit</th>

        @endif

        <th>Delete</th>

      </tr>
    </thead>

    <tr>
      <td>
```

```
    {{ $profile->id }}
```

```
</td>
```

```
<td>
```

```
    <a href="/profile/{{ $profile->id }}/edit">
```

```
        {{ $profile->fullName() }}
```

```
    </a>
```

```
</td>
```

```
<td>
```

```
    {{ $profile->showGender($profile->gender) }}
```

```
</td>
```

```
<td>
```

```
    {{ $profile->birthdate->format('m-d-Y') }}
```

```
</td>
```

```
@if(Auth::user()->adminOrCurrentUserOwns($profile))
```

```
<td>
```

```
    <a href="/profile/{{ $profile->id }}/edit">
```

```
        <button type="button"
                class="btn btn-default">
```

```
            Edit
```

```
        </button>
```

```
    </a>
```

```
</td>
```

```
@endif
```

```
<td>
    <div class="form-group">

        <form class="form"
              role="form"
              method="POST"
              action="{{ url('/profile/'. $profile->id) }}>

            <input type="hidden" name="_method" value="delete">

            {{ csrf_field() }}

            <input class="btn btn-danger"
                  Onclick="return ConfirmDelete();"
                  type="submit"
                  value="Delete">

        </form>

    </div>
</td>

</tr>
</tbody>

</table>

</div>

@endsection

@section('scripts')

<script>

    function ConfirmDelete()
    {

        var x = confirm("Are you sure you want to delete?");
        return x;
    }

</script>
```

```
    }  
  
</script>  
  
@endsection
```

So most of this is not new to us. But we do have a check in the breadcrumbs to see if the user is admin or not. The reason is that if they are admin, they get a link to all profiles:

```
'Profile' => '/profile'
```

If they are not admin, they do not get that link.

Index View - Profile

index.blade.php

From book:

```
@extends('layouts.master')  
  
@section('title')  
  
    <title>Profiles</title>  
  
@endsection  
  
@section('content')  
  
    <ol class='breadcrumb'>  
        <li><a href='/'>Home</a></li>  
        <li class='active'>Profiles</li>  
    </ol>
```

```
<h2>Profiles</h2>

<hr/>

@if($profiles->count() > 0)






```

So on our index page, we are creating a list of profiles, just like we did for widgets. Let's move on to the edit view.

Edit View - Profile

Gist:

[edit.blade.php](#)

From book:

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

@section('title')
    <title>Edit a Profile</title>
@endsection

@section('content')
    @if(Auth::user()->isAdmin())
        <ol class='breadcrumb'>
            <li><a href='/'>Home</a></li>
            <li><a href='/profile'>Profiles</a></li>
            <li><a href='/profile/{{ $profile->id }}'>
                {{ $profile->fullName() }}
            </a></li>
        </ol>
    @else
        <ol class='breadcrumb'>
            <li><a href='/'>Home</a></li>
            <li><a href='/profile/{{ $profile->id }}'>
                {{ $profile->fullName() }}
            </a></li>
        </ol>
    @endif

```

```
@endif

<h2>Edit Your Profile</h2>

<hr/>

<form class="form"
      role="form"
      method="POST"
      action="{{ url('/profile/'. $profile->id) }}>

  <input type="hidden" name="_method" value="patch">

  {{ csrf_field() }}

  <!-- first_name Form Input -->

  <div class="

    form-group{{ $errors->has('first_name') ? ' has-error' : '' }}>

    " >

    <label class="control-label">First Name</label>

    <input type="text"
          class="form-control"
          name="first_name"
          value="{{ $profile->first_name }}>

    @if ($errors->has('first_name'))

      <span class="help-block">

        <strong>{{ $errors->first('first_name') }}</strong>

      </span>

    @endif

  </div>

  <!-- last_name Form Input -->
```

```
<div class="form-group{{ $errors->has('last_name') ? ' has-error' : '' }}>

    <label class="control-label">Last Name</label>

    <input type="text"
           class="form-control"
           name="last_name"
           value="{{ $profile->last_name }}>

    @if ($errors->has('last_name'))

        <span class="help-block">
            <strong>{{ $errors->first('last_name') }}</strong>
        </span>

    @endif

</div>

<!-- birthdate Form Input -->

<div class="form-group{{ $errors->has('birthdate') ? ' has-error' : '' }}>

    <label class="control-label">Birthdate</label>

    <div>
        {{ Form::date('birthdate', $profile->birthdate) }}
    </div>

    @if ($errors->has('birthdate'))
```

```
<span class="help-block">

<strong>{{ $errors->first('birthdate') }}</strong>

</span>

@endif

</div>

<!-- Gender Form Input -->

<div class="

form-group{{ $errors->has('gender') ? ' has-error' : '' }}>

">

<label class="control-label">Gender</label>

<select class="form-control" id="gender" name="gender">

<option value="{{ $profile->showGender($profile->gender) }}>

{{ $profile->showGender($profile->gender) }}
```

```
@endif

</div>

<div class="form-group">

    <button type="submit" class="btn btn-primary btn-lg">

        Update

    </button>

</div>

</form>

@endsection
```

There's nothing really new to discuss here. You can see we are using \$profile->birthdate to give us the old value of the profile record:

```
{{ Form::date('birthdate', $profile->birthdate) }}
```

Ok, so you can play around with this and create your profile. You'll see that when you click the profile link from the dropdown, if you don't have a profile, it goes to the create view, otherwise, it goes to the show view.

Users For Admin

In the course of administering your application, you will want to have access to user data. Since all passwords are encrypted, you will not have access to that private information.

But you will be able to modify users, for example, assigning is_admin to a user, so they can reach the backend. You will also be able to delete users.

We already have the user model, so we don't need to create the model or migration. So let's start with the user routes:

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

Next we need our controller:

```
php artisan make:controller UserController --resource
```

UserController.php

Gist:

[UserController.php](#)

From book:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Requests\UserRequest;
use DB;
use App\User;
use Redirect;

class UserController extends Controller
{

    public function __construct()
    {

        $this->middleware(['auth', 'admin']);

    }

    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
}
```

```
public function index()
{
    $users = User::paginate(10);

    return view('user.index', compact('users'));

}

/**
 * Display the specified resource.
 *
 * @param int $id
 * @return \Illuminate\Http\Response
 */

public function show($id)
{
    $user = User::findOrFail($id);

    $profile = $user->profile;

    return view('user.show', compact('user', 'profile'));
}

/**
 * Show the form for editing the specified resource.
 *
 * @param int $id
 * @return \Illuminate\Http\Response
 */

public function edit($id)
{
    $user = User::findOrFail($id);

    return view('user.edit', compact('user'));
}
```

```
/**  
 * Update the specified resource in storage.  
 *  
 * @param \Illuminate\Http\Request $request  
 * @param int $id  
 * @return \Illuminate\Http\Response  
 */  
  
public function update(UserRequest $request, $id)  
{  
  
    $user = User::findOrFail($id);  
  
    $user->updateUser($user, $request);  
  
    alert()->success('Congrats!', 'You updated a user');  
  
    return Redirect::route('user.show', ['user' => $user]);  
}  
  
/**  
 * Remove the specified resource from storage.  
 *  
 * @param int $id  
 * @return \Illuminate\Http\Response  
 */  
  
public function destroy($id)  
{  
  
    User::destroy($id);  
  
    alert()->overlay(  
        'Attention!', 'You deleted a user', 'error'  
    );  
  
    return Redirect::route('user.index');  
}  
}
```

You can see we are enforcing admin and auth middleware on all methods:

```
$this->middleware(['auth', 'admin']);
```

This is because this controller is meant for Admin only. We will create a separate route and controller for the user to interact with their user record and we will call that settings. But for now let's discuss our User controller.

In the show method, we are creating the profile object from the relationship:

```
$profile = $user->profile;
```

I don't know how much easier it could get to work with than that. It's just simply beautiful. That's why it's aptly named Eloquent.

On the update method, we have something new. Instead of Request type-hinting \$request, we have:

```
UserRequest $request
```

This is our own custom request class. We haven't built that yet, we're going to do it in a minute. But that file will need to check if the authenticated user is the owner of the record or is admin. This is similar to a method that we have in our OwnsRecord trait, but that is not exactly what we need. So let's add the following method to our OwnsRecord trait:

Gist:

```
allowUserUpdate
```

From book:

```
public function allowUserUpdate($user)
{
    if (Auth::user()->isAdmin()){

        return true;

    }

    return $user->id === Auth::id();

}
```

Ok, now we're ready to make our Request class.

UserRequest

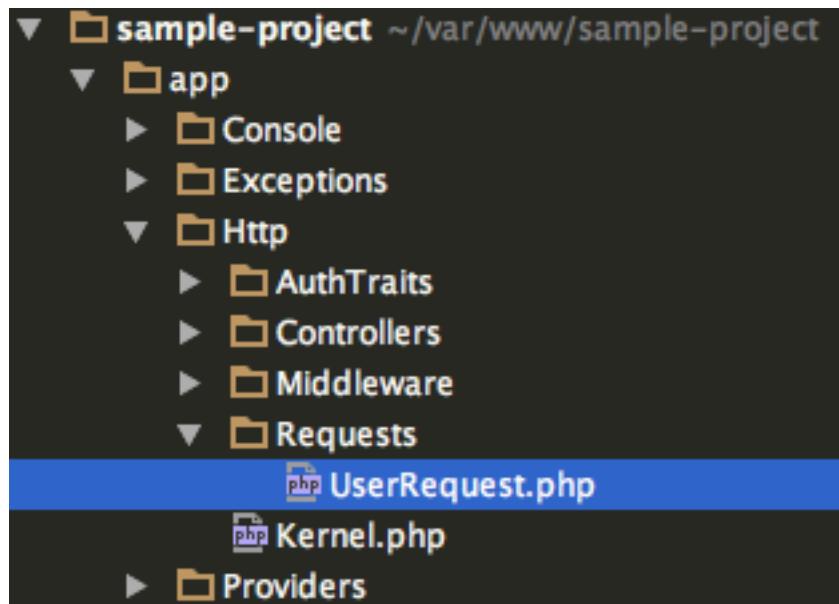
Laravel really helps you create all of these specialized classes. Not only will it create the class for us, but it will put in the proper folder for us as well.

Let's run this from the command line:

```
php artisan make:request UserRequest
```

You can see the convention we are following, uppercase first letter of the words, last word is Request. This will not only make the file for us, but since it's the first Request class, it will also make the Requests folder for us.

The new file will appear in app/Http/Requests:



Go ahead and browse the empty stub, then change the contents of UserRequest.php to the following.

Gist:

[UserRequest.php](#)

From book:

```
<?php

namespace App\Http\Requests;

use App\Http\Auth\Traits\OwnsRecord;
use Illuminate\Foundation\Http\FormRequest;

class UserRequest extends FormRequest
{
    use OwnsRecord;
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */

    public function authorize()
```

```

{
    if ( ! $this->allowUserUpdate($this->user)) {

        return false;
    }

    return true;
}

/**
 * Get the validation rules that apply to the request.
 *
 * @return array
 */

public function rules()
{
    return [
        'name' => 'required|string|max:20|unique:users,name,' .
            $this->user,
        'email' => 'required|email|unique:users,email,' . $this->user,
        'is_subscribed' => 'boolean',
        'is_admin' => 'boolean',
        'status_id' => 'in:7,10',
    ];
}
}

```

Please use the Gist for formatting. Note there should be no spaces in between the pipe characters, having spaces can cause problems.

So all we've done is add a call to our allowUserUpdate method and move the validation rules to their own class, which cleans up the controller a bit.

We've seen the unique rule before. It's a little tricky because you have to account for the update method, and unique will fail there, unless you append . \$this->user.

Note that the syntax is a little different from doing the same thing in the controller, where it would be . \$id. In this case, we bring in the local instance of the wildcard on the route, which we have to access via \$this.

Also in the authorize method we are using `$this->user`, which is coming in from the wildcard on the route. If you want, you can run:

```
php artisan route:list
```

You will see on the update route the `{user}` wildcard. I hope that makes sense.

Back to the UserController, you might be wondering where are the create and store methods? We don't need those, since users create their records when they register.

Ok, so the next thing we see that is different in our update method is we use the following:

```
$user->updateUser($user, $request);
```

The `updateUser` method is a new method we will be placing on our User Model. The purpose of this is to extract out logic from the controller to make it less bulky and easier to read. The controller method is very concise:

```
public function update(UserRequest $request, $id)
{
    $user = User::findOrFail($id);

    $user->updateUser($user, $request);

    alert()->success('Congrats!', 'You updated a user');

    return Redirect::route('user.show', ['user' => $user]);
}
```

Ok, since we're talking about `updateUser`, it's a perfect transition into working on the model.

User Model changes

To make sure we get the complete User model, I'm going to give you the whole thing, then review the new parts:

Gist:

[User.php](#)

From book:

```
<?php

namespace App;

use App\Http\Auth\Traits\OwnsRecord;
use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Support\Facades\Auth;
use App\Http\Requests\UserRequest;

class User extends Authenticatable
{
    use Notifiable, OwnsRecord;

    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = [
        'name',
        'email',
        'is_subscribed',
        'is_admin',
        'status_id',
        'password',
    ];

    /**
     * The attributes that should be hidden for arrays.
     *
     * @var array
     */
```

```
/*
protected $hidden = [
    'password',
    'remember_token',
];

public function isAdmin()
{
    return Auth::user()->is_admin == 1;
}

public function isActiveStatus()
{
    return Auth::user()->status_id == 10;
}

public function updateUser($user, UserRequest $request)
{
    return $user->update([
        'name' => $request->name,
        'email' => $request->email,
        'is_subscribed' => $request->is_subscribed,
        'is_admin' => $request->is_admin,
        'status_id' => $request->status_id,
    ]);
}

public function showAdminStatusOf($user)
{
    return $user->is_admin ? 'Yes' : 'No';
}
```

```
public function showNewsletterStatusOf($user)
{
    return $user->is_subscribed == 1 ? 'Yes' : 'No';
}

public function widgets()
{
    return $this->hasMany('App\Widget');
}

public function socialProviders()
{
    return $this->hasMany('App\SocialProvider');
}

public function profile()
{
    return $this->hasOne('App\Profile');
}

}
```

Let's look at what we have that is new on the model.

We were just talking about the updateUser method:

```
public function updateUser($user, UserRequest $request)
{
    return $user->update([
        'name' => $request->name,
        'email' => $request->email,
        'is_subscribed' => $request->is_subscribed,
        'is_admin' => $request->is_admin,
        'status_id' => $request->status_id,
    ]);
}
```

This is a simple method that simply uses Eloquent's update method to set the column names to the given request values. Since we are using UserRequest, we have to pull that in as a use statement:

```
use App\Http\Requests\UserRequest;
```

Next we have some methods that format our attributes for display. The first one is showAdminStatusOf:

```
public function showAdminStatusOf($user)
{
    return $user->is_admin ? 'Yes' : 'No';
}
```

We use a simple ternary to tell us what to display.

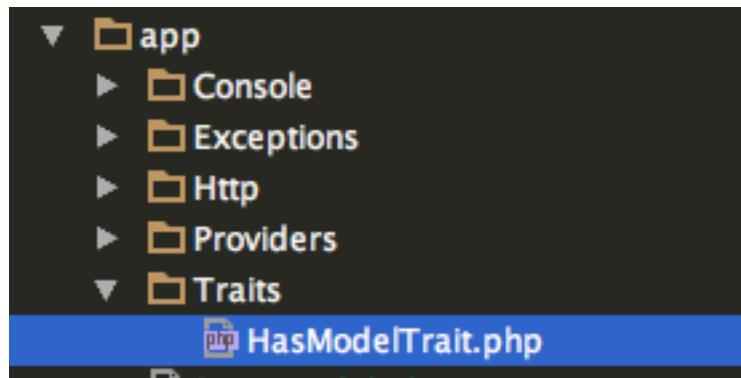
We have something similar for the next method:

```
public function showNewsletterStatusOf($user)
{
    return $user->is_subscribed == 1 ? 'Yes' : 'No';
}
```

Now we could have added the showStatusOf method to the model, but we're going to add it to HasModelTrait instead. The reason for this is that different models may need to have a status formatter and there is no reason to do it twice. We could add it to SuperModel, but User doesn't extend off of that model, so, we'll just update HasModelTrait.php. We ended up having a purpose for that after all.

HasModelTrait

If for any reason you chopped out the Traits directory and the HasModelTrait.php file, go ahead and create them again:



Remove the accessor from HasModelTrait, since we have that in SuperModel.php and add the following method to HasModelTrait.php.

Gist:

[showStatusOf](#)

From book:

```
public function showStatusOf($record)
{
    switch ($record) {
        case $record->status_id == 10:
            return 'Active';
            break;
        case $record->status_id == 7:
            return 'Inactive';
            break;
        default:
            return 'Inactive';
    }
}
```

Here is a gist of the entire file:

[HasModelTrait.php](#)

So if we wanted a ‘Pending’ status, for example, we could just add it as case statement. That makes this super-easy to work with.

Ok, let’s add the use statement at the top of User.php:

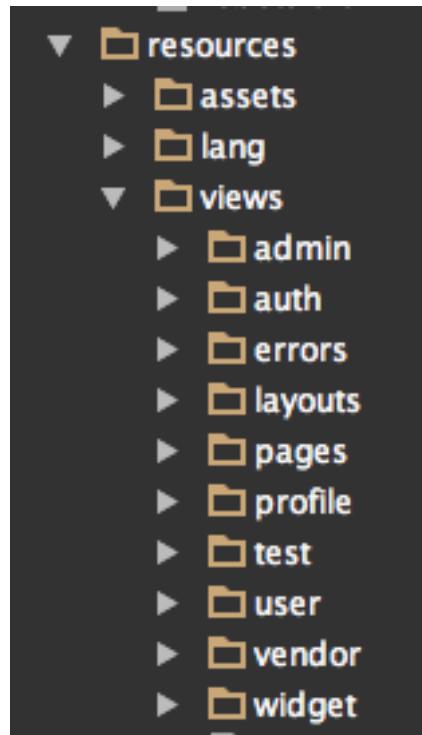
```
use App\Http\Auth\Traits\OwnsRecord;
use App\Traits\HasModelTrait;
use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Support\Facades\Auth;
use App\Http\Requests\UserRequest;

class User extends Authenticatable
{
    use Notifiable, OwnsRecord, HasModelTrait;
```

Here is a gist of the entire file for reference:

User.php

Now let's make a user folder in views:



Ok, so now we're ready for our views. I'm providing all the code for the views for those working offline, however there isn't much to discuss, since we have covered this ground previously. So we'll move through it quickly.

Index View

Gist:

Index View

From book:

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

@section('title')
    <title>Users</title>

@endsection

@section('content')
    <ol class='breadcrumb'>
        <li><a href='/'>Home</a></li>
        <li class='active'>Users</li>
    </ol>

    <h2>Users</h2>

    <hr/>

    @if($users->count() > 0)

        <table class="table table-hover table-bordered table-striped">

            <thead>
                <th>Id</th>
                <th>Name</th>
                <th>Admin</th>
                <th>Status</th>
                <th>Newsletter</th>
                <th>Date Created</th>
                <th>Edit</th>
                <th>Delete</th>
            </thead>

            <tbody>

                @foreach($users as $user)
```

```
<tr>
    <td>{{ $user->id }}</td>
    <td><a href="/user/{{ $user->id }}">
        {{ $user->name }}</a></td>
    <td>{{ $user->showAdminStatusOf($user) }}</td>
    <td>{{ $user->showStatusOf($user) }}</td>
    <td>{{ $user->showNewsletterStatusOf($user) }}</td>
    <td>{{ $user->created_at->format('m-d-Y') }}</td>

    <td><a href="/user/{{ $user->id }}/edit">
        <button type="button"
            class="btn btn-default">
            Edit
        </button></a></td>

    <td>
        <div class="form-group">

            <form class="form"
                role="form"
                method="POST"
                action="{{ url('/user/'. $user->id) }}>

                <input type="hidden" name="_method" value="delete">
                {{ csrf_field() }}

                <input class="btn btn-danger"
                    Onclick="return ConfirmDelete();"
                    type="submit" value="Delete">

            </form>
        </div>
    </td>
</tr>

@endforeach

</tbody>
```

```

        </table>

@else

    Sorry, no Users

@endif

{{ $users->links() }}

@endsection

```

Use the gist for formatting.

Show View

Gist:

[show.blade.php](#)

From book:

```

@extends('layouts.master')

@section('title')
    <title>{{ $user->name }}</title>
@endsection

@section('content')

@if(Auth::user()->isAdmin())

    <ol class='breadcrumb'>
        <li><a href='/'>Home</a></li>
        <li><a href='/user'>Users</a></li>
        <li><a href='/user/{{ $user->id }}'>
            {{ $user->name }}</a></li>

```

```
</ol>

@else

<ol class='breadcrumb'>
    <li><a href='/'>Home</a></li>
    <li><a href='/user/{{ $user->id }}'>
        {{ $user->name }}</a></li>
</ol>

@endif

<br>

<h1>User: {{ $user->name }}</h1>

<hr/>

<div class="panel panel-default">

    <!-- Default panel contents -->

    <!-- Table -->

    <table class="table table-striped">

        <thead>
            <tr>

                <th>Id</th>
                <th>Name</th>
                <th>Profile</th>
                <th>Email</th>
                <th>Subscribed</th>
                <th>Admin</th>
                <th>Status</th>
                <th>Created</th>
                @if(Auth::user()->adminOrCurrentUserOwns($user))
                    <th>Edit</th>
                @endif
                <th>Delete</th>

            </tr>
        </thead>
```

```
<tbody>
<tr>
    <td>{{ $user->id }} </td>

    <td> <a href="/user/{{ $user->id }}/edit">
        {{ $user->name }}</a></td>

@if (isset($profile->id))

    <td> <a href="/profile/{{ $profile->id }}">
        {{$user->profile->fullName()}}</a></td>
@else

    <td>none</td>

@endif

    <td>{{ $user->email }}</td>
    <td>{{ $user->showNewsletterStatusOf($user) }}</td>
    <td>{{ $user->showAdminStatusOf($user) }}</td>
    <td>{{ $user->showStatusOf($user) }}</td>
    <td>{{ $user->created_at->format('m-d-Y') }}</td>

@if(Auth::user()->adminOrCurrentUserOwns($user))

    <td> <a href="/user/{{ $user->id }}/edit">
        <button type="button"
            class="btn btn-default">
            Edit
        </button></a></td>

@endif

<td>
    <div class="form-group">

        <form class="form"
            role="form"
            method="POST"
```

```
        action="{{ url('/user/'. $user->id) }}>

<input type="hidden" name="_method" value="delete">

{{ csrf_field() }}

<input class="btn btn-danger"
       Onclick="return ConfirmDelete();"
       type="submit"
       value="Delete">

</form>
</div>
</td>
</tr>

</tbody>

</table>

</div>

@endsection

@section('scripts')

<script>

function ConfirmDelete()
{
    var x = confirm("Are you sure you want to delete?");
    return x;
}

</script>

@endsection
```

We check to see if the user has a profile, if so link to it, if not show static text:

```
@if (isset($profile->id))

<td> <a href="/profile/{{ $profile->id }}">
{{ $user->profile->fullName() }}</a></td>

@else

<td>none</td>

@endif
```

We are also using a number of our model and trait methods for formatting:

```
<td>{{ $user->showNewsletterStatusOf($user) }}</td>
<td>{{ $user->showAdminStatusOf($user) }}</td>
<td>{{ $user->showStatusOf($user) }}</td>
<td>{{ $user->created_at->format('m-d-Y') }}</td>
```

So we are putting all the methods we created to good use.

Edit View

Gist:

[edit.blade.php](#)

From book:

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

@section('title')
    <title>Edit a User</title>

@endsection

@section('content')
    @if(Auth::user()->isAdmin())
        <ol class='breadcrumb'>
            <li><a href='/'>Home</a></li>
            <li><a href='/user'>Users</a></li>
            <li><a href='/user/{{ $user->id }}'>
                {{ $user->name }}</a></li>
        </ol>
    @else
        <ol class='breadcrumb'>
            <li><a href='/'>Home</a></li>
            <li><a href='/user/{{ $user->id }}'>
                {{ $user->name }}</a></li>
        </ol>
    @endif
    <h2>Edit Your Record</h2>
    <hr/>
    <form class="form"
          role="form"
          method="POST"
          action="{{ url('/user/'. $user->id) }}">
        <input type="hidden" name="_method" value="patch">
        {{ csrf_field() }}
        <!-- first_name Form Input -->
```

```
<div class="

    form-group{{ $errors->has('name') ? ' has-error' : '' }}

">

<label class="control-label">First Name</label>

<input type="text"
       class="form-control"
       name="name"
       value="{{ $user->name }}>

@if ($errors->has('name'))

<span class="help-block">

<strong>{{ $errors->first('name') }}</strong>

</span>

@endif

</div>

<!-- is_admin Form Input -->

<div class="

    form-group{{ $errors->has('is_admin') ? ' has-error' : '' }}

">

<label class="control-label">Is Admin?</label>

<select class="form-control"
        id="is_admin"
        name="is_admin">

<option value="{{ $user->showAdminStatusOf($user) }}>

{{ $user->showAdminStatusOf($user) }}>
```

```
</option>

<option value="1">Yes</option>

<option value="0">No</option>

</select>

@if ($errors->has('is_admin'))

<span class="help-block">

<strong>{{ $errors->first('is_admin') }}</strong>

</span>

@endif

</div>

<div class="form-group">

<button type="submit" class="btn btn-primary btn-lg">

    Update

</button>

</div>

</form>

@endsection
```

There are a number of dropdowns, but we have gone over this before and there isn't anything new to discuss in our edit view.

Navigation to Users & Profiles

Ok, so now we need a way to access our users and profiles as admin, which means we will get the index views with the lists of users and profiles.

To do this, we are going to make a change to nav.blade.php. Here is the entire file.

Gist:

[nav.blade.php](#)

From book:

```
<!-- Fixed navbar -->

<nav class="navbar navbar-inverse navbar-fixed-top">

  <div class="container">

    <div class="navbar-header">

      <button type="button"
              class="navbar-toggle collapsed"
              data-toggle="collapse"
              data-target="#navbar" aria-expanded="false"
              aria-controls="navbar">

        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>

      </button>
      <a class="navbar-brand"
          href="/">

        Sample Project

      </a>

    </div>

    <div id="navbar" class="navbar-collapse collapse pull-right">
```

```
<ul class="nav navbar-nav">

    <li class="active"><a href="/">Home</a></li>

    <li><a href="#about">About</a></li>

    @if (Auth::check() && Auth::user()->isAdmin())

        <li class="dropdown">

            <a href="#" 
                class="dropdown-toggle" 
                data-toggle="dropdown" 
                role="button" 
                aria-haspopup="true" 
                aria-expanded="false">

                Users

                <span class="caret"></span>

            </a>

            <ul class="dropdown-menu">

                <li><a href="/user">Users</a></li>

                <li><a href="/profile">Profiles</a></li>

            </ul>

        </li>

    @endif

    <li class="dropdown">

        <a href="#" 
            class="dropdown-toggle" 
            data-toggle="dropdown" 
            role="button"
```

```
        aria-haspopup="true"
        aria-expanded="false">

    Content

    <span class="caret"></span>

    </a>
    <ul class="dropdown-menu">

        <li><a href="/widget">Widgets</a></li>

    </ul>

</li>

@if (Auth::check())

<li class="dropdown">

    <a href="#"
        class="dropdown-toggle"
        data-toggle="dropdown"
        role="button"
        aria-haspopup="true"
        aria-expanded="false">

        {{ Auth::user()->name }}</a>

    <span class="caret"></span>

    </a>

    <ul class="dropdown-menu">

        <li><a href="/determine-profile-route">

            Profile

        </a>

    </li>

    </ul>

</li>
```

```
</li>
<li><a href="/auth/facebook">
    <i class="fa fa-facebook"></i>

    &nbsp;&nbsp;
    Facebook Sync

</a>
</li>

<li><a href="/auth/github">
    <i class="fa fa-github"></i>

    &nbsp;&nbsp;
    Github Sync

</a>
</li>

<li>
    <a href="/logout"
        onclick="event.preventDefault();
        document.getElementById('logout-form')
        .submit();">

        Logout

    </a>

    <form id="logout-form"
        action="/logout"
        method="POST"
        style="display: none;">

        {{ csrf_field() }}

    </form>

</li>

</ul>

</li>
```

```

        src="{{ Gravatar::get(Auth::user()->email) }}">
    </li>
@else
    <li><a href="/login">Login</a></li>
    <li><a href="/register">Register</a></li>
    <li>
        <a href="/auth/facebook">
            <i class="fa fa-facebook"></i>

            &nbsp;&nbsp;
            Sign in </a>
    </li>
    <li><a href="/auth/github">
        <i class="fa fa-github"></i>

        &nbsp;&nbsp;
        Sign in</a>
    </li>
</ul>

@endif
</div><!--/.nav-collapse -->
</div>

</nav>

```

Please note I have to break the `` for the Gravatar into two lines here to avoid wordwrap, but that needs to be one line or else it will break.

So what we did is add the following block:

```

@if (Auth::check() && Auth::user()->isAdmin())

```

- <li class="dropdown">

 Users

 <ul class="dropdown-menu">
 Users
 Profiles

 @endif

In the above, I double-spaced it, so you can see it more clearly. We test to see if the user is logged in and admin, and if so, they get to see the dropdown named Users that includes links to users and profiles.

Go ahead and test these links, they should return lists of users and profiles respectively.

I also added links to sync facebook and github as well as a sign in link for Github.

Admittedly, the top nav is a bit of a Frankenstein at this point. It's looking cluttered. I will leave it up to you to ultimately decide exactly what you wish to include there.

Settings

So we have our admin view of users in place, but we need a way for individual users to access and modify their user records. We won't need any new models or migrations for this, but we will need 2 routes, a controller, and a view.

Settings Routes

```
Route::get('settings', 'SettingsController@edit');

Route::post('settings', 'SettingsController@update')
    ->name('user-update');
```

Now let's make a settings controller:

```
php artisan make:controller SettingsController
```

We wanted a plain controller, so we left off the `-resource` flag.

SettingsController

Only 2 methods on this one.

Gist:

[SettingsController.php](#)

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\User;
use Illuminate\Support\Facades\Auth;
use Redirect;

class SettingsController extends Controller
{

    public function __construct()
    {
```

```
$this->middleware('auth');
```

```
}
```

```
/**
```

```
 * Show the form for editing the specified resource.
```

```
*
```

```
* @param int $id
```

```
* @return Response
```

```
*/
```

```
public function edit()
```

```
{
```

```
    $id = Auth::user()->id;
```

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

```
    return view('settings.edit', compact('user'));
```

```
}
```

```
/**
```

```
 * Update the specified resource in storage.
```

```
*
```

```
* @param Request $request
```

```
* @param int $id
```

```
* @return Response
```

```
*/
```

```
public function update(Request $request)
```

```
{
```

```
    $id = Auth::user()->id;
```

```
    $this->validate($request, [
```

```
        'name' => 'required|max:20',
```

```
        'email' => 'required|email|max:255|unique:users,email,' . $id,
```

```
        'is_subscribed' => 'boolean'
```

```
    ]);
```

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

```
    $user->update([ 'name' => $request->name,
```

```
                  'email' => $request->email,
```

```

        'is_subscribed' => $request->is_subscribed]);
    alert()->success('Congrats!', 'You updated your user settings');

    return redirect()->action('SettingsController@edit', [$user]);
}

}

```

So the edit method will only return the edit form for the Auth user:

```

public function edit()
{
    $id = Auth::user()->id;
    $user = User::findOrFail($id);

    return view('settings.edit', compact('user'));
}

```

We get the correct user and compact the user and send to the view. Then when we update via the update method:

```

public function update(Request $request)
{
    $id = Auth::user()->id;

    $this->validate($request, [
        'name' => 'required|max:20',
        'email' => 'required|email|max:255|unique:users,email,' .
        '$id',
        'is_subscribed' => 'boolean'
    ]);
}

```

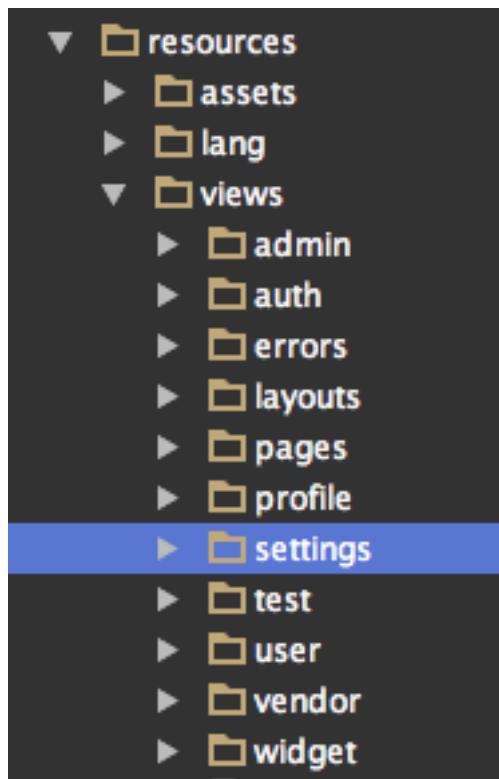
```
]);  
  
$user = User::findOrFail($id);  
  
$user->update(['name' => $request->name,  
               'email' => $request->email]);  
  
alert()->success('Congrats!', 'You updated your user settings');  
  
return redirect()->action('SettingsController@edit', [$user]);  
  
}
```

If they want to update their password, we force them to use the forgot password link, which is on the edit view. That way it offers a validation method, which we already happen to have written for us, so that works out really well.

Also note that we are redirecting to a controller action, and passing in the \$user object, so that's a helpful option for redirect.

Create Settings View Folder

Ok, it's time to work on our view. We need to create the settings folder within the views folder. It will look like this:



Edit View For Settings

Next we need to create an edit.blade.php file within the settings folder with the following contents.

Gist:

[edit.blade.php](#)

From book:

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

@section('title')
    <title>Edit Your User Settings</title>

@endsection

@section('content')
```

```
<ol class="breadcrumb">
    <li><a href="/">Home</a></li>
    <li>Settings</li>
</ol>

<div class="pull-right">
    <a href="{{ url('/password/reset') }}">
        <button type="button"
                class="btn btn-lg btn-primary">
            Reset Password
        </button>
    </a>
</div>

<h1 class="myTableFont">Update {{ $user->name }}</h1>
<hr/>

<form class="form"
      role="form"
      method="POST"
      action="{{ url('/settings') }}">
    {{ csrf_field() }}

    <!-- name Form Input -->

    <div class="form-group{{ $errors->has('name') ? ' has-error' : '' }}>
        <label class="control-label">User Name</label>
        <input type="text"
```

```
        class="form-control"
        name="name"
        value="{{ $user->name }}"

@if ($errors->has('name'))

<span class="help-block">

<strong>{{ $errors->first('name') }}</strong>

</span>

@endif

</div>

<div class="

form-group{{ $errors->has('email') ? ' has-error' : '' }}>

<label for="email" class="control-label">

E-Mail Address

</label>

<div>

<input id="email"
      type="email"
      class="form-control"
      name="email"
      value="{{ $user->email }}>

@if ($errors->has('email'))

<span class="help-block">

<strong>{{ $errors->first('email') }}</strong>

</span>
```

```
@endif

</div>
</div>

<!-- is_subscribed Form Input -->

<div class="

    form-group{{ $errors->has('is_subscribed') ? ' has-error' : '' }}"

>

<label class="control-label">Is Subscribed?</label>

<select class="form-control"
        id="is_subscribed"
        name="is_subscribed">

<option value="{{ $user->is_subscribed }}>
    {{ $user->is_subscribed == 1 ? 'Yes' : 'No' }}</option>

<option value="1">Yes</option>
<option value="0">No</option>
</select>

@if ($errors->has('is_subscribed'))

<span class="help-block">
    <strong>{{ $errors->first('is_subscribed') }}</strong>
</span>
@endif

</div>
```

```
<div class="form-group">

    <button type="submit"
        class="btn btn-primary btn-lg">
        Update
    </button>

</div>

</form>

@endsection
```

Reminder, please Gist for formatting the code.

You can see we are using the forgot password link and button:

```
<div class="pull-right">

<a href="/password/email">

    <button type="button"
        class="btn btn-lg btn-primary">
        Reset Password
    </button>

</a>

</div>
```

In order for that to work, however, we are going to have to modify the middleware on our ForgotPassword-Controller to the following:

```
$this->middleware('guest', ['except'=> [
    'showLinkRequestForm',
    'sendResetLinkEmail'
]]);
```

The reason that we need this is that we see the settings edit page after being logged in. So if we didn't make the exceptions, we would be redirected to the login view because of the 'guest' middleware. Alternatively, we could just remove the middleware, but I thought it was safer just to list the exceptions.

Similarly, we have to modify the ResetPasswordController:

```
protected $redirectTo = '/';

public function __construct()
{
    $this->middleware('guest', ['except'=> [
        'showResetForm',
        'reset'
]]);
}
```

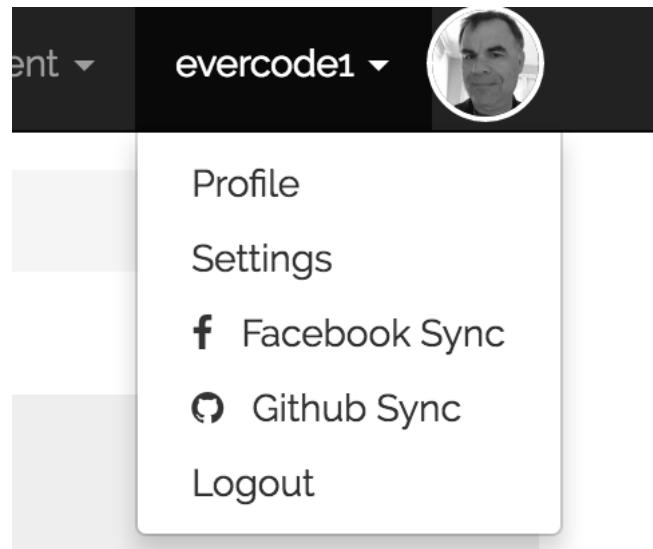
You'll note that I added the redirectTo property as well. We need this for the reset to work because we removed the /home route early in the book.

Add Settings To Nav

To finish off our settings, we need to add it to our nav, it's just one line that goes below our Profile link:

```
<li><a href="/settings">Settings</a></li>
```

It will look like this, when you are done:



Wow, I really need a new gravatar photo....

Anyway, you can see we have a nicely formatted dropdown list for our users.

I will provide the gist of the entire nav.blade.php file, in case anyone needs to reference it.

Gist:

[nav.blade.php](#)

Admin Page

Before we conclude the chapter, let's add some content to our admin area. Create a file named grid.blade.php in our views/admin folder with the following contents:

Gist:

[grid.blade.php](#)(<https://gist.github.com/evercode1/ee3f27c42406463251e9>)

From book:

```
<div class="row">
<div class="col-xs-6 col-lg-4">
<a href="/user"><h2>Users</h2></a>
<a href="/user">
<p>

Use this link to manage your applications's
users

</p>
</a>
<p><a class="btn btn-default"
    href="/user"
    role="button">View details &raquo;</a></p>
</div><!--/.col-xs-6.col-lg-4-->

<div class="col-xs-6 col-lg-4">
<a href="/profile"><h2>Profiles</h2></a>
<a href="/profile">
<p>

Use this link to manage your applications's profiles

</p>
</a>
<p><a class="btn btn-default"
    href="/profile"
    role="button">View details &raquo;</a></p>
</div><!--/.col-xs-6.col-lg-4-->

<div class="col-xs-6 col-lg-4">
<h2>Heading</h2>
<p>Donec id elit non mi porta gravida at eget metus. Fusce dapibus,
    tellus ac cursus commodo,tortor mauris condimentum nibh, ut
    fermentum massa justo sit amet risus. Etiam porta sem malesuada
    magna mollis euismod. Donec sed odio dui. </p>
<p><a class="btn btn-default" href="#"
    role="button">View details &raquo;</a></p>
</div><!--/.col-xs-6.col-lg-4-->

<div class="col-xs-6 col-lg-4">
<h2>Heading</h2>
<p>Donec id elit non mi porta gravida at eget metus. Fusce dapibus,
    tellus ac cursus commodo,tortor mauris condimentum nibh, ut
```

```
fermentum massa justo sit amet risus. Etiam porta sem malesuada magna mollis euismod. Donec sed odio dui. </p>
<p><a class="btn btn-default" href="#" role="button">View details &raquo;</a></p>
</div><!--/.col-xs-6.col-lg-4-->

<div class="col-xs-6 col-lg-4">
<h2>Heading</h2>
<p>Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo,tortor mauris condimentum nibh, ut fermentum massa justo sit amet risus. Etiam porta sem malesuada magna mollis euismod. Donec sed odio dui. </p>
<p><a class="btn btn-default" href="#" role="button">View details &raquo;</a></p>
</div><!--/.col-xs-6.col-lg-4-->

<div class="col-xs-6 col-lg-4">
<h2>Heading</h2>
<p>Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo,tortor mauris condimentum nibh, ut fermentum massa justo sit amet risus. Etiam porta sem malesuada magna mollis euismod. Donec sed odio dui. </p>
<p><a class="btn btn-default" href="#" role="button">View details &raquo;</a></p>
</div><!--/.col-xs-6.col-lg-4-->

</div><!--/row-->
```

Next we need to modify admin/index.blade.php to the following:

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

@section('title')

<title>The Admin Page</title>

@endsection

@section('content')
```

```
<h1>Admin</h1>

@include('admin.grid')

@endsection
```

We don't need a gist, you are only changing the `<h1>` and adding a single line:

```
<h1>Admin</h1>

@include('admin.grid')
```

So when you are done, it looks like this:

Sample Project

Home About Users Content gumby

Admin

Users

Use this link to manage your applications's users

[View details >](#)

Profiles

Use this link to manage your applications's profiles

[View details >](#)

Heading

Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo, tortor mauris condimentum nibh, ut fermentum massa justo sit amet risus. Etiam porta sem malesuada magna mollis euismod. Donec sed odio dui.

[View details >](#)

Heading

Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo, tortor mauris condimentum nibh, ut fermentum massa justo sit amet risus. Etiam porta sem malesuada magna mollis euismod. Donec sed odio dui.

[View details >](#)

Heading

Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo, tortor mauris condimentum nibh, ut fermentum massa justo sit amet risus. Etiam porta sem malesuada magna mollis euismod. Donec sed odio dui.

[View details >](#)

© 2015 - 2017 Sample Project All rights Reserved.

Ok, so you can see this is the beginnings of a very basic admin panel. Our sample project keeps style to a minimum, it is up to you to decorate it as you wish.

Let's add a link to admin from our dropdown list. In the dropdown list under our username, where we have Profile and Logout, let's add the following as the first list item:

```
@if(Auth::user()->isAdmin())  
  
<li><a href="/admin">Admin</a></li>  
  
@endif
```

Now we have a convenient way to get to our admin page if we are admin.

Add Facebook sign in buttons to Login and Register Views

While we're doing a bit of clean up, let's add a Facebook button to our Login and Register views, so our ui presents a more consistent experience.

In both register.blade.php and login.blade.php, we will add the following, just below the div class="col-md-8 col-md-offset-2.

Gist:

[facebook button](#)

From book:

```
<div>  
  
<a href="/auth/facebook">  
  
    <button type="button"  
        class="btn btn-primary btn-lg btn-block">  
  
        Facebook Sign In  
  
    </button>  
  
</a>  
  
</div>  
  
<div>Login in with your Facebook sign in above or use the form below:</div>
```

Obviously on the register view you will replace the word Login with Register.

I will provide the gist for both files for reference:

[login.blade.php](#)

[register.blade.php](#)

Your page should look like this:

The screenshot shows a registration form titled "Facebook Sign In". The form is part of a "Sample Project" website, as indicated by the header. The navigation bar includes links for Home, About, Content, Login, Register, and Sign in. The current page is "Register", as shown in the breadcrumb trail. The form itself has a blue header and contains fields for Name, E-Mail Address, Password, and Confirm Password. There are also checkboxes for "Subscribe to Newsletter?" and "Agree To Terms", both of which are checked. A "Register" button is at the bottom of the form. At the bottom of the page, there is a footer with the text "© 2015 - 2017 Sample Project All rights Reserved".

If you don't like the button spanning the length of the form, feel free to adjust how you wish. We are keeping our template css as basic as possible.

I didn't put Github on these pages, so obviously, this is more of a starting point than a finished UI. But since UI is a not a strong point for me, I can leave it to you to take it from here.

View Composers

Sometimes we need to send data to the view without hitting a controller first, and typically this is done through view partials where you want to include the view partial and its data into several other pages.

A really trivial example of this could be our layouts.bottom page, which is included in every page call via the master page.

Right now, we have the following PHP in that view:

```
<hr>

<div class="well">

<p>&copy;

@if (date('Y') > 2015)
2015 - {{ date('Y') }}

@else

2015

@endif

Sample Project All rights Reserved.</p>

</div>
```

There isn't much to it, and depending on your project size, this may be fine. However we can improve it by extracting out the logic, so that it ultimately looks like this:

```
<hr>

<div class="well">

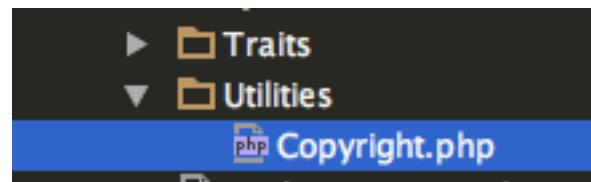
<p> {!! $copyright !!}</p>

</div>
```

Right now that doesn't work because we don't have a controller handing a \$copyright variable. And obviously, since layouts/bottom.blade.php is included on every page, passing it from every controller on the site is not a good option.

We solve this by creating a view composer. In order to build the view composer, we have to extract out the logic we want to a class, so we can call it and send it to the view.

In our app folder, let's create a Utilities folder, and within that create a file named Copyright.php.



The Utilities folder is in the app folder and will appear below the Traits folder.

Next, let's change the contents of the Copyright.php file.

Gist:

[Copyright.php](#)

From book

```
<?php

namespace App\Utilities;

class Copyright
{

    public static function displayNotice()
    {

        $date = date('Y') > 2015 ? date ('Y') : 2015;

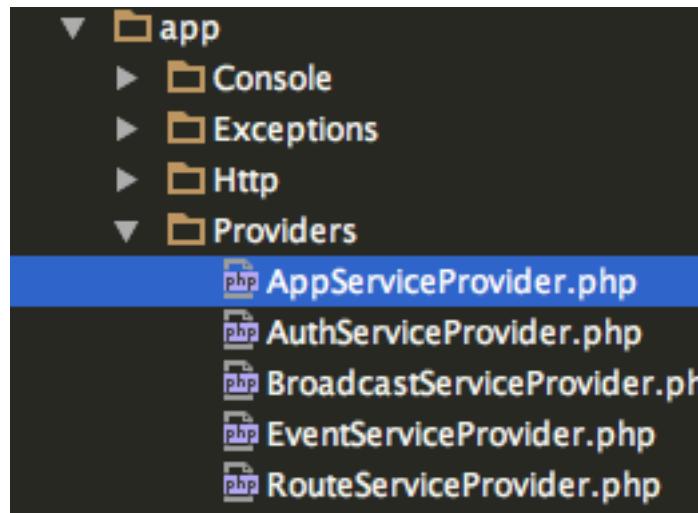
        return '&copy ' . $date .
            ' Sample Project All rights Reserved.';

    }

}
```

So we've created a simple Copyright class, namespaced under AppUtilities. It has one static function named displayNotice that returns a string showing the copyright year and copyright notice. The '©' string is what gives the c with a circle around it.

Next will open our app/Providers/AppServiceProvider.php file, located here:



You can see that you got two methods that are empty stubs. Let's change the boot method to the following:

Gist:

[boot method](#)

From book:

```
public function boot()
{
    view()->composer('layouts.bottom', function($view){
        $view->with('copyright',
            \App\Utilities\Copyright::displayNotice());
    });
}
```

The first parameter of the composer method is the view you want to bind to, in this case, ‘layouts.bottom.’ The second parameter is a closure with the view variable passed in. We use the with() method to hand in two arguments, the first is what you want to name the variable. In this case we are calling it ‘copyright’, so in the view we will access it as \$copyright. The second argument is the call to the method in the Copyright class, where we supply the full namespace.

Since we’re focused on view composers, I won’t go too deep on service providers. If you want more info, you can check out the [service provider docs](#).

Finally, let’s modify layouts/bottom.blade.php to the following:

```
<hr>

<div class="well">

    <p> {!! $copyright !!}</p>

</div>
```

Note that in the blade syntax we are using different opening and closing tags, {!! !!}. That’s how you have to do it when you don’t want to escape html entities, which is what we want in this case because of the © entity for the copyright symbol.

View Share

If you want to share the value of a variable in all views, you could do it as follows:

Gist: [view share](#)

From book:

```
public function boot()
{
    $value = \App\Utilities\Copyright::displayNotice();

    view()->share('copyright', $value);
}
```

In this case, the share method first parameter is the key and the second is the value.

Another way to make modular data content for your application is with ajax calls and Vue components. We will be exploring that in detail in chapter 12.

Summary

Congratulations, you now have a basic sample project in laravel that supports both the front-end and back-end. Users can login and register via Facebook and create their profiles.

You have a basic idea of how to stand up a model with a RESTful pattern, like we have done with the widgets model. That makes a great point of reference for future projects.

Chapter 10: Working With Images

Working with images is a bit of pain. We're so used to working with nice, neat data models that working with files seems so much more cumbersome.

Laravel doesn't completely alleviate the headache, but it does have some intuitive syntax and there is a really helpful package named [Intervention Image](#) that we can pull in to help us.

We're not going to only learn how to manage images, but we're also going to build a responsive bootstrap carousel that will populate the marketing images dynamically.

This will give potential clients a lot of control over the UI, which they will appreciate. Instead of having to call you every time they want to add an image or change the order of the images, they will be able to edit things like that themselves. Clients love having that kind of control and they will appreciate your attention to detail.

Ok, so let's pull in our Imagine package by running the following from the command line:

```
composer require intervention/image
```

Please note that you can't update config/app before composer is done installing the package, it will cause an error.

After the package is installed, we need to make our package visible to the application by adding to the providers array in config/app.php:

```
Intervention\Image\ImageServiceProvider::class,
```

And in the same file in the aliases array:

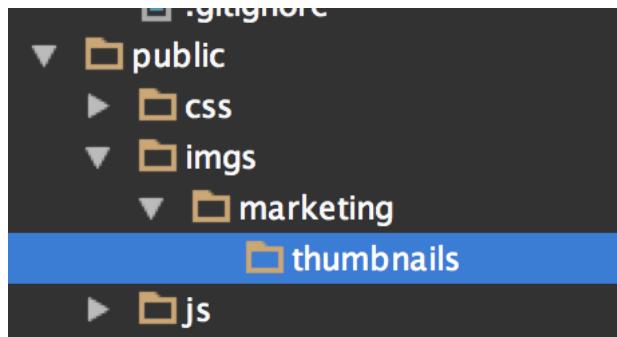
```
'Image'      => Intervention\Image\Facades\Image::class,
```

And with that we have the package successfully pulled in.

We can think of the images that we are going to use for the carousel as marketing images. We will of course need a place to store them.

Create imgs, marketing-images, and thumbnails folder.

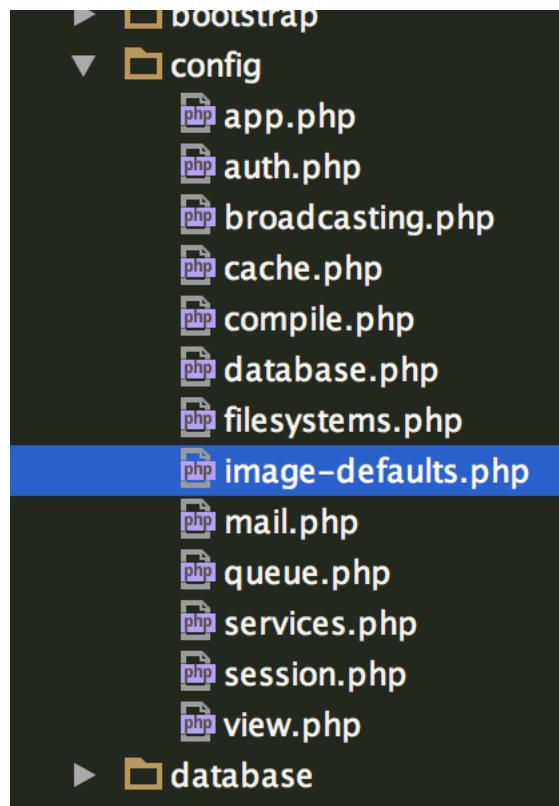
Let's create an imgs folder within our public folder. Then make a marketing folder within that and a thumbnails folder within the marketing folder. It should look like this:



It's worth noting that Laravel integrates well Amazon's S3 and other non-local file-systems. That's more of an advanced topic, so we won't be covering it. If you want to read up on those features, check out:

[file systems](#)

For our purposes, the imgs folder inside the public folder will work fine. And now that we have that, we are going to set up a configuration file to hold the paths of our images. Go to your config folder and create `image-defaults.php`:



Place the following contents in image-defaults.php:

Gist:

[image-defaults](#)

From book:

```
<?php
```

```
return [
```

```
/*
|--------------------------------------------------------------------------
| Default Image Paths and Settings
|--------------------------------------------------------------------------
|
|
|
| We set the config here so that we can keep our controllers clean.
| Configure each image type with an image path.
|
```

```
*/  
  
'marketingImage' => [  
  
    'destinationFolder'      => '/imgs/marketing/',  
    'destinationThumbnail'   => '/imgs/marketing/thumbnails/',  
    'thumbPrefix'            => 'thumb-',  
    'imagePath'              => '/imgs/marketing/',  
    'thumbnailPath'          => '/imgs/marketing/thumbnails/thumb-',  
    'thumbHeight'            => 60,  
    'thumbWidth'             => 60,  
  
,  
  
];
```

You can see it's an array that holds the paths and other values that we need. Now to call these values, we use something like this:

```
$imageDefaults = Config::get('image-defaults.marketingImage');
```

The syntax is simple, we use the get method of the Config class to reference the image-defaults file and assign the values to \$imageDefaults.

If we had several keys within the image-defaults config file, let's say you had profile images and widget images, etc. you could hand in the key through a variable like so:

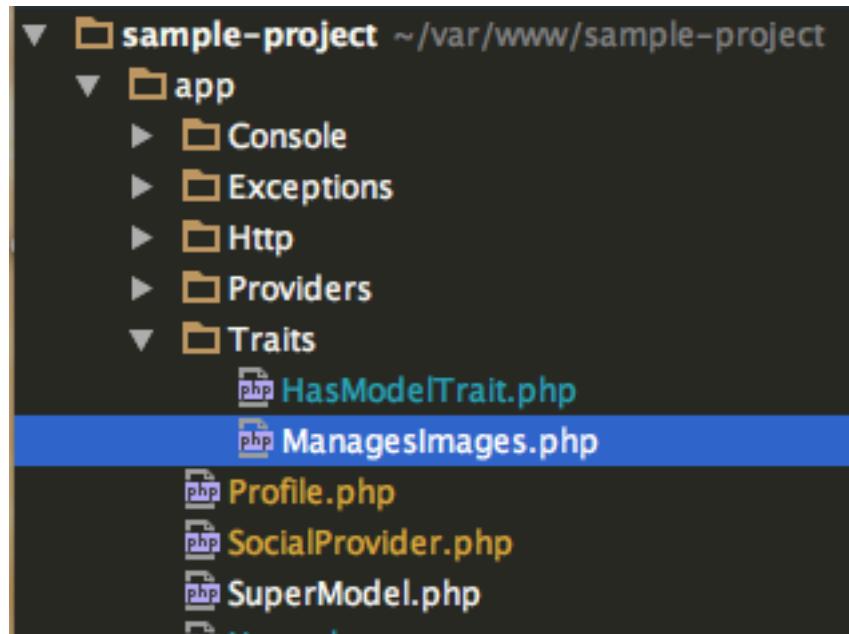
```
$imageTypeKey = 'marketingImage';  
  
$imageType = 'image-defaults.' . $imageTypeKey;  
  
$imageDefaults = Config::get($imageType);
```

You would of course have to have corresponding values in the image-defaults.php config file if you wanted to use this for other image types.

To make the marketing image paths available to whichever controllers or models may need them, I'm going to create a trait. The trait will also contain reusable methods for other models, should we choose to use it to manage more image models.

When we name traits, we follow the convention of what the trait does, so I think `ManagesImages` would be a good name for it.

In your traits folder, create `ManagesImages.php`:



`ManagesImages.php` should be as follows:

Gist:

[ManagesImages](#)

From book:

```
<?php
namespace App\traits;

use Illuminate\Support\Facades\Config;
use Intervention\Image\Facades\Image;
use Illuminate\Http\UploadedFile;
use Illuminate\Support\Facades\Input;
use Illuminate\Support\Facades\File;

trait ManagesImages
```

```
{  
  
    public $destinationFolder;  
    public $destinationThumbnail;  
    public $extension;  
    public $file;  
    public $imageDefaults;  
    public $imageName;  
    public $imagePath;  
    public $thumbHeight;  
    public $thumbPrefix;  
    public $thumbnailPath;  
    public $thumbWidth;  
  
    /**  
     * @param $modelImage  
     * hand in the model  
     */  
  
    private function deleteExistingImages($modelImage)  
    {  
        // delete old images before saving new  
  
        $this->deleteImage($modelImage, $this->destinationFolder);  
  
        $this->deleteThumbnail  
            ($modelImage, $this->destinationThumbnail);  
  
    }  
  
    Private function deleteImage($modelImage, $destination)  
    {  
  
        File::delete(public_path($destination) .  
            $modelImage->image_name . '.' .  
            $modelImage->image_extension);  
    }  
  
    Private function deleteThumbnail($modelImage, $destination)  
    {  
  
        File::delete(public_path($destination) . $this->thumbPrefix .  
            $modelImage->image_name . '.' .
```

```
$modelImage->image_extension);  
}  
  
private function getUploadedFile()  
{  
  
    return $file = Input::file('image');  
  
}  
  
private function makeImageAndThumbnail()  
{  
    //create instance of image from temp upload  
  
    $image = Image::make($this->file->getRealPath());  
  
    //save image with thumbnail  
  
    $image->save(public_path()  
  
        . $this->destinationFolder  
        . $this->imageName  
        . '.'  
        . $this->extension)  
    ->resize($this->thumbWidth, $this->thumbHeight)  
  
    // ->greyscale()  
  
    ->save(public_path()  
        . $this->destinationThumbnail  
        . $this->thumbPrefix  
        . $this->imageName  
        . '.'  
        . $this->extension);  
  
}  
  
/**  
 * @return bool  
 */  
  
private function newFileIsUploaded()
```

```
{  
  
    return !empty(Input::file('image'));  
  
}  
  
private function saveImageFiles(UploadedFile $file, $model)  
{  
  
    $this->setImageFile($file);  
  
    $this->setFileAttributes($model);  
  
    $this->makeImageAndThumbnail();  
  
}  
  
private function setImageDefaultsFromConfig($imageTypeKey)  
{  
  
    $imageType = 'image-defaults.' . $imageTypeKey;  
  
    $this->imageDefaults = Config::get($imageType);  
  
    $this->setImageProperties();  
  
}  
  
private function setFileAttributes($model)  
{  
  
    $this->imageName = $model->image_name;  
    $this->extension = $model->image_extension;  
  
}  
  
private function setImageProperties()  
{  
  
    foreach ($this->imageDefaults as  
            $propertyName => $PropertyValue){  
  
        if ( property_exists( $this , $propertyName ) {  
  
    }  
}
```

```
$this->$propertyName = $propertyValue;  
}  
  
}  
  
}  
  
private function setImageFile(UploadedFile $file)  
{  
  
    $this->file = $file;  
  
}  
  
}
```

Ok, so this has a bunch of properties and 11 methods. We'll start by looking at two methods and then we will come back to the trait as we use it in workflow.

Let's look at:

```
private function setImageDefaultsFromConfig($imageTypeKey)  
{  
    $imageType = 'image-defaults.' . $imageTypeKey;  
  
    $this->imageDefaults = Config::get($imageType);  
  
    $this->setImageProperties();  
}
```

As we discussed earlier, we can pull the image paths and other defaults from the config file, which hold them in an array \$this->imageDefaults. Then we call the second method to iterate through the rest of the properties and set them:

```
private function setImageProperties()
{
    foreach ($this->imageDefaults as
        $propertyName => $propertyValue){

        if ( property_exists( $this , $propertyName ) ){

            $this->$propertyName = $propertyValue;

        }
    }
}
```

Obviously I put the foreach on two lines to avoid word-wrap.

The setImageProperties() method is a handy helper to pull all the values from the array and set them as class properties. We include the check:

```
if ( property_exists( $this , $propertyName ) {
```

So this way we will only try to set property values that have already been declared. Running this method will set the following properties:

```
public $destinationFolder;
public $destinationThumbnail;
public $extension;
public $imageDefaults;
public $imagePath;
public $thumbHeight;
public $thumbPrefix;
public $thumbnailPath;
public $thumbWidth;
```

Obviously, we can't set the file and imageName properties until we consume the request on the controller.

Don't worry if you don't quite get how it works now, we will demonstrate all of this shortly. The important thing to get is that this will be a reusable trait, so if we have any other controllers that do image management, we should be able to use the trait to help us.

Ok, moving on, we're going to create a model to associate and control the images.

Let's create a MarketingImage model with migration, from the command line:

```
php artisan make:model MarketingImage -m
```

Let's change the boilerplate up method to:

Gist:

[up method](#)

From book:

```
public function up()
{
    Schema::create('marketing_images', function(Blueprint $table)
    {

        $table->increments('id');
        $table->boolean('is_active')->default(false);
        $table->boolean('is_featured')->default(false);
        $table->string('image_name')->unique();
        $table->string('image_extension', 10);
        $table->timestamps();
    });
}
```

```
});  
}
```

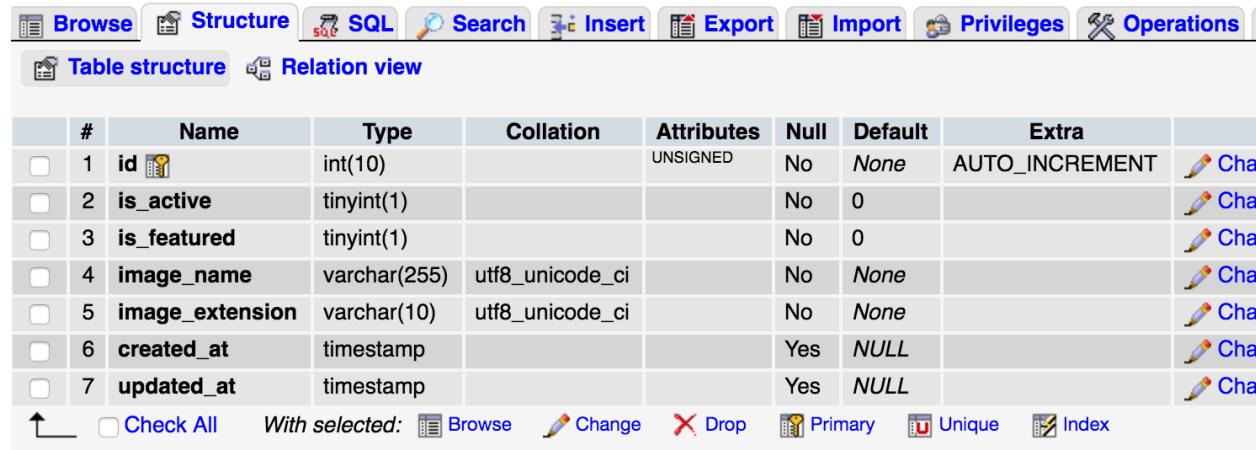
So you can see what we're planning to do. We will save the names of our images along with a couple of boolean columns that determine if they are active or featured. Active means they will be included in the carousel, featured means that it is the first slide.

For convenience, we will also save the file extension, so we will always have that available to us when we want to use the image.

Ok, from the command line, let's run:

```
php artisan migrate
```

So your PhpAdmin table should look like this:



	#	Name	Type	Collation	Attributes	Null	Default	Extra
<input type="checkbox"/>	1	id	int(10)		UNSIGNED	No	<i>None</i>	AUTO_INCREMENT
<input type="checkbox"/>	2	is_active	tinyint(1)			No	0	
<input type="checkbox"/>	3	is_featured	tinyint(1)			No	0	
<input type="checkbox"/>	4	image_name	varchar(255)	utf8_unicode_ci		No	<i>None</i>	
<input type="checkbox"/>	5	image_extension	varchar(10)	utf8_unicode_ci		No	<i>None</i>	
<input type="checkbox"/>	6	created_at	timestamp			Yes	<i>NULL</i>	
<input type="checkbox"/>	7	updated_at	timestamp			Yes	<i>NULL</i>	

Check All With selected:

Since we're going to want to use the accessor we created for the created_at date format, we will extend SuperModel, instead of Model, so let's make that change:

```
class MarketingImage extends SuperModel
```

Since SuperModel lives in the same namespace as MarketingImage, we do not need a use statement.

Next, let's add the fillable columns to our MarketingImage model:

```
protected $fillable = ['is_active',
                      'is_featured',
                      'image_name',
                      'image_extension'];
```

Next let's create a route resource for MarketingImage:

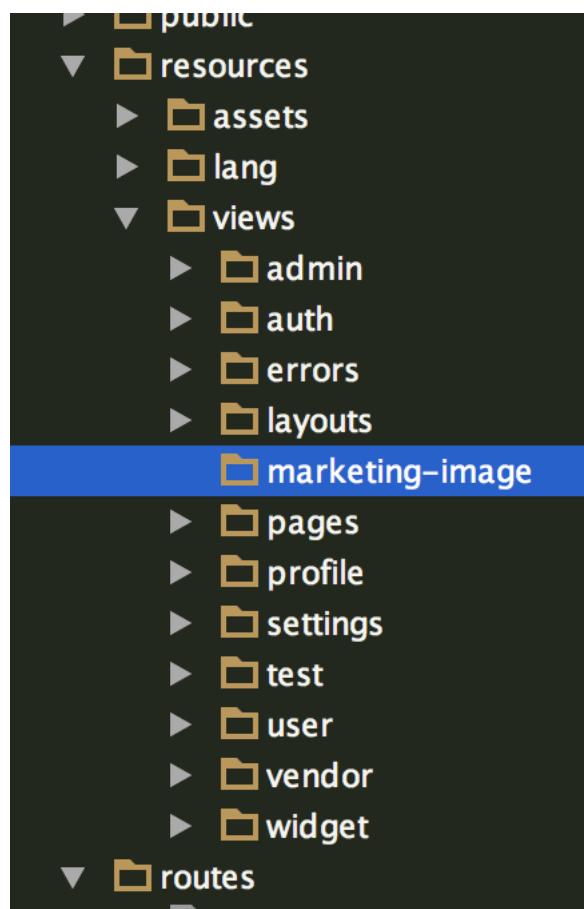
```
Route::resource('marketing-image', 'MarketingImageController');
```

You can see we are breaking up the word in the route, using a dash between the words. We are referencing the MarketingImageController that we don't have yet.

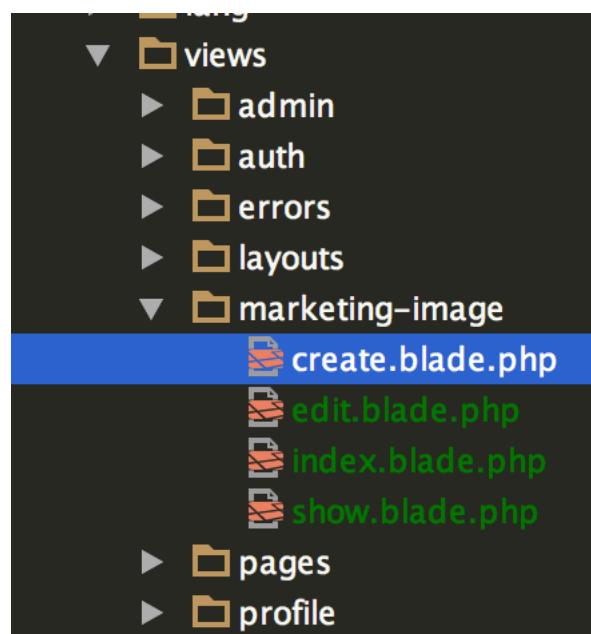
Let's make that now:

```
php artisan make:controller MarketingImageController --resource
```

So you can see we are following the pattern we established when we built our Widget model. And since we are doing that, the next step would be to make the marketing-image folder in views:



Now, as a shortcut, I'm simply going to copy all the files in the widget view folder into the marketing-image folder to use them as a starting point. It looks like this:



You might not need screenshots at this point, but on the other hand, they help make things crystal clear, so I will keep including them.

So obviously, we will start with the create form. Let's open up our `create.blade.php` and change it to the following:

Gist:

[create.blade.php](#)

From book:

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

@section('title')

<title>Create a Marketing Image</title>

@endsection

@section('content')

<ol class='breadcrumb'>
    <li><a href='/'>Home</a></li>
    <li><a href='/marketing-image'>Marketing Images</a></li>
    <li class='active'>Create</li>
```

```
</ol>

<h2>Create a New Marketing Image</h2>

<hr/>

<form class="form"
      role="form"
      method="POST"
      action="{{ url('/marketing-image') }}"
      enctype="multipart/form-data">

    {{ csrf_field() }}

    <!-- image_name Form Input -->

    <div class="

        form-group{{ $errors->has('image_name') ? ' has-error' : '' }}>

        <label class="control-label">Image Name</label>

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

        @if ($errors->has('image_name'))

            <span class="help-block">

                <strong>{{ $errors->first('image_name') }}</strong>

            </span>

        @endif

    </div>

    <!-- is_active Form Input -->

    <div class="
```

```
form-group{{ $errors->has('is_active') ? ' has-error' : '' }}

">

<label class="control-label">Is Active</label>

<select class="form-control"
        id="is_active"
        name="is_active">

    <option value="{{ old('is_active') }}>
        {{ ! is_null(old('is_active')) ? (old('is_active') == 1 ? 'Yes' : 'No') : 'Please Choose One' }}</option>

    <option value="1">Yes</option>
    <option value="0">No</option>

</select>

@if ($errors->has('is_active'))

    <span class="help-block">
        <strong>{{ $errors->first('is_active') }}</strong>
    </span>
@endif

</div>

<!-- is_featured Form Input --&gt;

&lt;div class="

    form-group{{ $errors-&gt;has('is_featured') ? ' has-error' : '' }}&gt;

"&gt;</pre>
```

```
<label class="control-label">Is Featured</label>

<select class="form-control"
        id="is_featured"
        name="is_featured">

    <option value="{{old('is_featured')}}">
        {{ ! is_null(old('is_featured')) ? (old('is_featured') == 1 ? 'Yes' : 'No') : 'Please Choose One' }}</option>

    <option value="1">Yes</option>
    <option value="0">No</option>

</select>

@if ($errors->has('is_featured'))

    <span class="help-block">
        <strong>{{ $errors->first('is_featured') }}</strong>
    </span>

@endif

</div>

<!-- image file Form Input --&gt;

&lt;div class="form-group{{ $errors-&gt;has('image') ? ' has-error' : '' }}&gt;

    &lt;div class="form-group"&gt;
        &lt;label class="control-label"&gt;</pre>
```

```
Primary Image

</label>

<input type="file"
       name="image"
       id="image">

</div>

@if ($errors->has('image'))

<span class="help-block">

<strong>{{ $errors->first('image') }}</strong>

</span>

@endif

<div class="form-group">

<button type="submit"
        class="btn btn-primary btn-lg">

    Create

</button>

</div>

</form>

@endsection
```

We can set the create method on the MarketingImageController to see the form:

```
public function create()
{
    return view('marketing-image.create');
}
```

That just simply returns the form, which should look like this:

The screenshot shows a web application interface. At the top, there is a dark header bar with the text "Sample Project" on the left and navigation links "Home", "About", "Users", "Content", and a user profile icon on the right. Below the header, the URL "Home / Marketing Images / Create" is displayed. The main content area has a title "Create a New Marketing Image". There are four input fields: "Image Name" (empty), "Is Active" (dropdown menu showing "Please Choose One"), "Is Featured" (dropdown menu showing "Please Choose One"), and "Primary Image" (file input field showing "Choose File No file chosen"). A blue "Create" button is located below these fields. At the bottom of the page, a footer bar contains the text "© 2015 - 2017 Sample Project All rights Reserved."

Back to the `create.blade` file, we do have some things that are different about this form. We start with the form opening:

```
<form class="form"
      role="form"
      method="POST"
      action="{{ url('/marketing-image') }}"
      enctype="multipart/form-data">

{{ csrf_field() }}
```

You can see that we have specified enctype. You need this in order to be able to upload a file.

For is_featured and is_active, we are going to use dropdown lists:

```
<!-- is_featured Form Input -->

<div class="form-group{{ $errors->has('is_featured') ? ' has-error' : '' }}>

<label class="control-label">Is Featured</label>

<select class="form-control"
        id="is_featured"
        name="is_featured">

<option value="{{ old('is_featured') }}>
    {{ ! is_null(old('is_featured')) ? (old('is_featured') == 1 ? 'Yes' : 'No') : 'Please Choose One' }}>

</option>

<option value="1">Yes</option>

<option value="0">No</option>

</select>
```

```
@if ($errors->has('is_featured'))  
  
    <span class="help-block">  
  
        <strong>{{ $errors->first('is_featured') }}</strong>  
  
    </span>  
  
@endif  
  
</div>
```

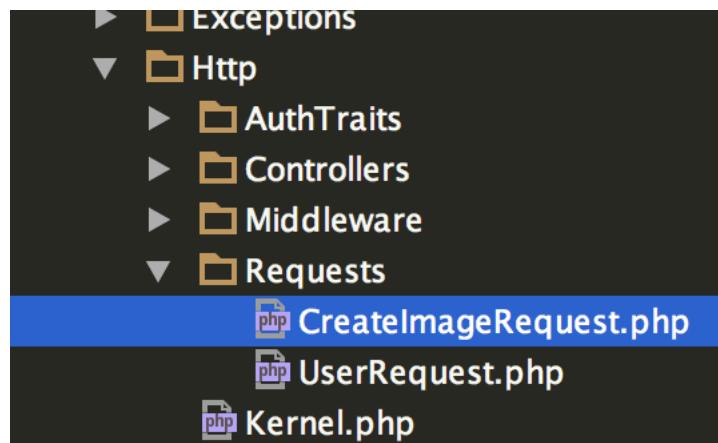
You can see we are using a nested ternary again to show the old form data or show “Please Choose One.” Then lastly, we can look at our file upload input:

```
<!-- image file Form Input -->  
  
<div class="  
  
    form-group{{ $errors->has('image') ? ' has-error' : '' }}  
  
">  
  
<div class="form-group">  
  
<label class="control-label">  
  
    Primary Image  
  
</label>  
  
<input type="file"  
       name="image"  
       id="image">  
  
</div>
```

Create Image Request

To validate the form input, we are going to create a request class to handle it. Let's make the request class. Form the command line run:

```
php artisan make:request CreateImageRequest
```



Now let's change the boilerplate to the following:

Gist:

[CreateImageRequest](#)

From book:

```
<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class CreateImageRequest extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
}
```

```
public function authorize()
{
    return true;
}

/**
 * Get the validation rules that apply to the request.
 *
 * @return array
 */

public function rules()
{
    return [
        'image_name' => 'alpha_num|required|unique:marketing_images',
        'is_active' => 'required|boolean',
        'is_featured' => 'required|boolean',
        'image' => 'required|mimes:jpeg,jpg,bmp,png|max:1000',
    ];
}
```

You can see the image is required and can have a max value of 1000 kb:

'image' => 'required|mimes:jpeg,jpg,bmp,png|max:1000',

Make sure in the rules not to have spaces before and after the pipe characters, they can sometimes cause issues.

Before I move on and give the full MarketingImageController, I want to show you what the store method on the controller used to look like before I built out the trait:

```
public function store(CreateImageRequest $request)
{
    //create new instance of model to save from form

    $marketingImage = new Marketingimage([
        'image_name'      => $request->get('image_name'),
        'image_extension' => $request->file('image')
                               ->getClientOriginalExtension(),
        'is_active'       => $request->get('is_active'),
        'is_featured'     => $request->get('is_featured'),
    ]);

    $marketingImage->save();

    //parts of the image we will need

    $file = Input::file('image');

    $imageName = $marketingImage->image_name;
    $extension = $request->file('image')
                           ->getClientOriginalExtension();

    //create instance of image from temp upload

    $image = Image::make($file->getRealPath());

    //save image with thumbnail

    $image->save(public_path() . $this->destinationFolder
                  . $imageName
                  . '.'
                  . $extension)
              ->resize(60, 60)

    // ->greyscale()

    ->save(public_path()
            . $this->destinationThumbnail
            . 'thumb-'
            . $imageName
            . '.'
            . $extension);
}
```

```
    alert()->success('Congrats!', 'Marketing Image Created!');

    return redirect()->route('marketing-image.show', [$marketingImage]);

}
```

It doesn't look very compact or friendly. The reason I'm mentioning it is that you will see a big difference in it when we have our methods extracted out to the trait, like so:

```
public function store(CreateImageRequest $request)
{
    //create new instance of model to save from form

    $marketingImage = new MarketingImage([
        'image_name'      => $request->get('image_name'),
        'image_extension' => $request->file('image')
                               ->getClientOriginalExtension(),
        'is_active'       => $request->get('is_active'),
        'is_featured'     => $request->get('is_featured')
    ]);

    // save model

    $marketingImage->save();

    // get instance of file

    $file = $this->getUploadedFile();

    // pass in the file and the model

    $this->saveImageFiles($file, $marketingImage);

    alert()->success(
        'Congrats!', 'Marketing Image And Thumbnail Created!'
    )
}
```

```
);

return redirect()->route('marketing-image.show', [$marketingImage]);

}
```

Ok, you can see this is much cleaner. We could still push the first two blocks of code onto the model, but leaving it like this helps us understand what is coming through the form. It's your call on that if you want to refactor it. Obviously you can see the rest of it is now very readable.

Ok, we have some work to do before we can step through this and make it all work. I'll give you the full MarketingImageController. If you are copying, copy from the gist, it will be formatted more sensibly, since I don't have to avoid word-wraps.

Gist:

[MarketingImageController](#)

From book:

```
<?php

namespace App\Http\Controllers;

use App\traits\ManagesImages;
use App\Http\Requests\CreateImageRequest;
use App\MarketingImage;
use App\Http\Requests>EditImageRequest;

class MarketingImageController extends Controller
{
    use ManagesImages;

    public function __construct()
    {
        $this->middleware('auth');
        $this->middleware('admin');
        $this->setImageDefaultsFromConfig('marketingImage');

    }
}
```

```
/**  
 * Display a listing of the resource.  
 *  
 * @return \Illuminate\Http\Response  
 */  
  
public function index()  
{  
    $thumbnailPath = $this->thumbnailPath;  
  
    $marketingImages = MarketingImage::paginate(10);  
  
    return view('marketing-image.index', compact(  
        'marketingImages',  
        'thumbnailPath'  
    ));  
  
}  
  
/**  
 * Show the form for creating a new resource.  
 *  
 * @return \Illuminate\Http\Response  
 */  
  
public function create()  
{  
    return view('marketing-image.create');  
}  
/**  
 * Store a newly created resource in storage.  
 *  
 * @param \Illuminate\Http\Request $request  
 * @return \Illuminate\Http\Response  
 */  
  
public function store(CreateImageRequest $request)  
{  
    //create new instance of model to save from form
```

```
$marketingImage = new MarketingImage([
    'image_name'      => $request->get('image_name'),
    'image_extension' => $request->file('image')
                           ->getClientOriginalExtension(),
    'is_active'       => $request->get('is_active'),
    'is_featured'     => $request->get('is_featured')
]);

// save model

$marketingImage->save();

// get instance of file

$file = $this->getUploadedFile();

// pass in the file and the model

$this->saveImageFiles($file, $marketingImage);

alert()->success(
    'Congrats!', 'Marketing Image And Thumbnail Created!'
);

return redirect()->route(
    'marketing-image.show', [$marketingImage]
);

}

/**
 * Display the specified resource.
 *
 * @param int $id
 * @return \Illuminate\Http\Response
 */

public function show($id)
```

```
{  
  
    $marketingImage = MarketingImage::findOrFail($id);  
  
    $thumbnailPath = $this->thumbnailPath;  
  
    $imagePath = $this-> imagePath;  
  
    return view('marketing-image.show', compact(  
  
        'marketingImage',  
        'thumbnailPath',  
        'imagePath'  
  
    ));  
  
}  
  
/**  
 * Show the form for editing the specified resource.  
 *  
 * @param int $id  
 * @return \Illuminate\Http\Response  
 */  
  
public function edit($id)  
{  
    $marketingImage = MarketingImage::findOrFail($id);  
  
    $thumbnailPath = $this->thumbnailPath;  
  
    return view('marketing-image.edit', compact(  
  
        'marketingImage',  
        'thumbnailPath'  
  
    ));  
  
}  
  
/**  
 * Update the specified resource in storage.  
 *  
 * @param \Illuminate\Http\Request $request  
 */
```

```
* @param int $id
* @return \Illuminate\Http\Response
*/
public function update($id, EditImageRequest $request)
{
    $marketingImage = MarketingImage::findOrFail($id);

    $this->setUpdatedModelValues($request, $marketingImage);

    // if file, we have additional requirements before saving

    if ($this->newFileIsUploaded()) {

        $this->deleteExistingImages($marketingImage);
        $this->setNewFileExtension($request, $marketingImage);

    }

    $marketingImage->save();

    // check for file, if new file, overwrite existing file

    if ($this->newFileIsUploaded()){

        $file = $this->getUploadedFile();
        $this->saveImageFiles($file, $marketingImage);

    }

    $thumbnailPath = $this->thumbnailPath;

    $imagePath = $this->imagePath;

    alert()->success('Congrats!', 'image edited!');

    return view('marketing-image.show', compact(
        'marketingImage',
        'thumbnailPath',
        'imagePath'
));
}
```

```
}

/**
 * Remove the specified resource from storage.
 *
 * @param int $id
 * @return \Illuminate\Http\Response
 */

public function destroy($id)
{
    $marketingImage = MarketingImage::findOrFail($id);

    $this->deleteExistingImages($marketingImage);

    MarketingImage::destroy($id);

    alert()->error('Notice', 'image deleted!');

    return redirect()->route('marketing-image.index');
}

/**
 * @param EditImageRequest $request
 * @param $marketingImage
 */
private function setNewFileExtension
    (EditImageRequest $request, $marketingImage)
{
    $marketingImage->image_extension = $request->file('image')
        ->getClientOriginalExtension();
}

/**
 * @param EditImageRequest $request
 * @param $marketingImage
 */
private function setUpdatedModelValues
```

```
(EditImageRequest $request, $marketingImage)
{
    $marketingImage->is_active = $request->get('is_active');
    $marketingImage->is_featured = $request->get('is_featured');

}
}
```

Please use the gist for code formatting.

Ok, we obviously pull in all the use statements that we need, including the use ManagesImages statement in the class to use the trait.

Then in the constructor, we call the setImageDefaultsFromConfig method:

```
$this->setImageDefaultsFromConfig('marketingImage');
```

So we already discussed how this method on the trait sets up our default configurations for our marketing images.

Let's return to the store method. You can see in the method signature that we are using our request class:

```
public function store(CreateImageRequest $request)
{
```

It's so cool the way Laravel handles its request objects. It helps keep the controller a little thinner as well.

If we fail validation, the process never makes into the method. If it passes, we get an instantiated instance named \$request to work with.

We make good use of that \$request instance in our store method on the MarketingImageController to help set the values of our new MarketingImage model:

```
//create new instance of model to save from form

$marketingImage = new Marketingimage([
    'image_name'      => $request->get('image_name'),
    'image_extension' => $request->file('image')
                           ->getClientOriginalExtension(),
    'is_active'       => $request->get('is_active'),
    'is_featured'     => $request->get('is_featured'),
]);


---


```

You can see that for the image_extension attribute, we use getClientOriginalExtension method to grab the extension from the file. I like storing this as a model attribute because it makes it easier to work with later on.

After creating it, we save the model with Eloquent's simple syntax:

```
$marketingImage->save();
```

Now the tricky thing to keep in mind when working with files is that they are distinct from the data model that we persisted to the DB, so we have to handle saving the image's physical file to the location we want separately.

We start with a trait method that gets us the file of the image we will need:

```
$file = $this->getUploadedFile();
```

On the trait, you can see getUploadedFile():

```
private function getUploadedFile()
{
    return $file = Input::file('image');

}
```

We use the input class to get the file with a name of image, which is the name we gave it in the create form on create.blade.php.

Now we could have just put directly into the method:

```
$file = Input::file('image');
```

That's actually less code. But it doesn't really tell you what it does, so we made a method to help us understand the code when we return to it a year from now.

Back on the store method on the controller, we call another trait method:

```
$this->saveImageFiles($file, $marketingImage);
```

That method is on the trait:

```
private function saveImageFiles(UploadedFile $file, $model)
{
    $this->setImageFile($file);
    $this->setFileAttributes($model);
    $this->makeImageAndThumbnail();
}
```

So that calls 3 other methods. Let's look at the first one:

```
private function setImageFile(UploadedFile $file)
{
    $this->file = $file;
}
```

Ok, that's not so hard to understand, it simply takes in the uploaded file and sets it as a property.

Next we have:

```
private function setFileAttributes($model)
{
    $this->imageName = $model->image_name;
    $this->extension = $model->image_extension;
}
```

Again, very simple stuff, we are setting properties to the model values.

Here is the makeImageAndThumbnail method:

```
private function makeImageAndThumbnail()
{
    //create instance of image from temp upload

    $image = Image::make($this->file->getRealPath());

    //save image with thumbnail

    $image->save(public_path()
        . $this->destinationFolder
```

```
    . $this->imageName
    . '.'
    . $this->extension)
->resize($this->thumbWidth, $this->thumbHeight)
// ->greyscale()
->save(public_path())
. $this->destinationThumbnail
. $this->thumbPrefix
. $this->imageName
. '.'
. $this->extension);

}
```

Here we are using the Image class from intervention. We start with creating an instance of the image from the temp upload:

```
$image = Image::make($this->file->getRealPath());
```

Then we call \$image->save with a bunch of chained methods to save the image and the thumbnail:

```
$image->save(public_path())

. $this->destinationFolder
. $this->imageName
. '.'
. $this->extension)

->resize($this->thumbWidth, $this->thumbHeight)
// ->greyscale()

->save(public_path() . $this->destinationThumbnail
. $this->thumbPrefix
. $this->imageName
. '.'
. $this->extension);
```

Notice that we are using the properties from the trait that were configured with defaults, such as `$this->destinationThumbnail`. Those are loaded in from the config file by the `setImageDefaultsFromConfig` method that gets called in the `MarketingImageController`.

You can also see we are defining the public path, using the `public_path()` method, concatenating the destination folder and the image name and extension, which are also properties configured from the defaults.

Next we chain the `resize` method for the thumbnails, in this case using more configuration properties. If want a different size thumbnail, go to `image-defaults.php` in the config folder and change it there.

I have the `greyscale` method commented out, but it's there for reference if you want to make your thumbnails greyscale.

Then finally, we chain another `save` method onto the same image instance, but this time it saves the thumbnail to the `thumbnail` folder, prepending `$this->thumbPrefix` onto the filename.

After that method finishes executing, back on the `MarketingImagesController`, we flash an alert to the session, and we redirect to the `show` route:

```
return redirect()->route('marketing-image.show', [$marketingImage]);
```

Note that when you redirect this way, we are passing along the model instance, and Laravel will automatically extract the id for us, so it knows which id to hand into the `show` method.

Show Method

```
public function show($id)
{
    $marketingImage = MarketingImage::findOrFail($id);

    $thumbnailPath = $this->thumbnailPath;

    $imagePath = $this->imagePath;

    return view('marketing-image.show', compact(
        'marketingImage',
        'thumbnailPath',
        'imagePath'
    ));
}
```

```
));  
}
```

Not much new there, we are just setting variables from the properties that were set from the config, so that we can pass them along to the view.

Add Display Methods to MarketingImage Model

Let's go ahead and add the following methods to the MarketingImage model:

Gist:

[display methods](#)

From book:

```
public function showActiveStatus($is_active)  
{  
  
    return $is_active == 1 ? 'Yes' : 'No';  
  
}  
  
public function showFeaturedStatus($is_featured)  
{  
  
    return $is_featured == 1 ? 'Yes' : 'No';  
  
}
```

You can see we are just giving ourselves a way to display these statuses as Yes or No.

ShowImages trait.

In your traits folder, make a file named ShowsImages.php with the following contents:

Gist:

[ShowsImages Trait](#)

From book:

```
<?php

namespace App\ Traits;

trait ShowsImages
{

    public function noCache()
    {

        return '?' . 'time=' . time();
    }

    public function showImage($imageModel, $path)
    {

        return $path . $imageModel->image_name
            .
            .
            .
        . $imageModel->image_extension
        . $this->noCache();

    }

}
```

The noCache method is a trick I learned for development, when you do not want to cache the images. In that case, you can put a get string at the end of the image name and the browser thinks it's unique. This is not good for performance, but really helps out when you're testing and you need to see the latest image.

Once you are done with development, you would remove that method from inside the showImage method.

The showImage method simply takes in the model and path to return a string that will return the image from the html tags:

```
public function showImage(MarketingImage $marketingImage, $path)
{
    return $path . $marketingImage->image_name
        . ' '
        . $marketingImage->image_extension
        . $this->noCache();
}
```

We will see that in action in a minute. First, let's modify the MarketingImage model to use the ShowsImages trait:

```
class MarketingImage extends SuperModel
{
    use ShowsImages;
```

Don't forget the use statement at the top:

```
use App\Traits\ShowsImages;
```

show view

Gist:

[show view](#)

From book:

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

@section('title')
    <title>{{ $marketingImage->name }}</title>

@endsection

@section('content')
    <ol class='breadcrumb'>
        <li><a href='/'>Home</a></li>
        <li><a href='/marketing-image'>Marketing Images</a></li>
        <li><a href='/marketing-image/{{ $marketingImage->id }}'>
            {{ $marketingImage->image_name }}</a></li>
    </ol>

    <h1>{{ $marketingImage->image_name }} Marketing Image</h1>

    <div class="pull-left">

        <a href="/marketing-image/{{ $marketingImage->id }}/edit">
            <button type="button"
                    class="btn btn-primary btn-lg">
                Edit Image
            </button></a>
    </div>

    <br>
    <br>

    <hr/>

    <div class="panel panel-default">
        <!-- Table -->
        <table class="table table-striped">
```

```
<tr>

<th>Thumbnail</th>

</tr>

<tr>

<td>



</td>

</tr>

<tr>

<th>Active?</th>

</tr>

<tr>

<td>{{ $marketingImage->showActiveStatus($marketingImage->is_active) }}</td>

</tr>

<tr>

<th>Featured?</th>

</tr>

<tr>

<td>{{ $marketingImage->showFeaturedStatus($marketingImage->is_featured) }}</td>

</tr>

<tr>

<th>Primary Image</th>
```

```
</tr>

<tr>

<td>

 }})
```

```
</div>

@endsection

@section('scripts')

<script>

    function ConfirmDelete()
    {

        var x = confirm("Are you sure you want to delete?");

        return x;
    }

</script>

@endsection
```

So obviously we have built a html table to hold our image and and it's attributes. We call the image like so:

```
<td>



<form class="form"
      role="form"
      method="POST"
      action="{{ url('/marketing-image/'. $marketingImage->id) }}">

<input type="hidden" name="_method" value="delete">

{{ csrf_field() }}

<input class="btn btn-danger"
       onclick="return ConfirmDelete();"
       type="submit"
       value="Delete">

</form>
</div>

<!-- image name no input -->

<div>

<div class="control-label">

Image Name:

</div>
```

```
<h4>{{ $marketingImage->image_name
    . .
    $marketingImage->image_extension }}>
</h4>

</div>

<div class="control-label">Thumbnail:</div>

<!-- image thumbnail -->

<div>



</div>

<br>

<form class="form"
      role="form"
      method="POST"
      action="{{ url('/marketing-image/' . $marketingImage->id) }}"
      enctype="multipart/form-data">

<input type="hidden" name="_method" value="patch">

{{ csrf_field() }}

<!-- is_active Form Input -->

<div class="

form-group{{ $errors->has('is_active') ? ' has-error' : '' }}>

<label class="control-label">Is Active</label>

<select class="form-control"
        id="is_active"
        name="is_active">

<option value="{{ $marketingImage->is_active }}>
```

```
    {{ $marketingImage->is_active == 1 ? 'Yes' : 'No' }}  
  
</option>  
  
<option value="1">Yes</option>  
  
<option value="0">No</option>  
  
</select>  
  
@if ($errors->has('is_active'))  
  
    <span class="help-block">  
  
        <strong>{{ $errors->first('is_active') }}</strong>  
  
    </span>  
  
@endif  
  
</div>  
  
<!-- is_featured Form Input -->  
  
<div class="  
  
    form-group{{ $errors->has('is_featured') ? ' has-error' : '' }}  
  
">  
  
<label class="control-label">Is Featured</label>  
  
<select class="form-control"  
    id="is_featured"  
    name="is_featured">  
  
<option value="{{ $marketingImage->is_featured }}>  
  
    {{ $marketingImage->is_featured == 1 ? 'Yes' : 'No' }}  
  
</option>
```

```
<option value="1">Yes</option>

<option value="0">No</option>

</select>

@if ($errors->has('is_featured'))

<span class="help-block">

<strong>{{ $errors->first('is_featured') }}</strong>

</span>

@endif

</div>

<!-- image file Form Input -->

<div class="

    form-group{{ $errors->has('image') ? ' has-error' : '' }}>

    <br>

    <div class="form-group">

        <label class="control-label">

            Primary Image

        </label>

        <input type="file" name="image" id="image">

    </div>

    @if ($errors->has('image'))

        <span class="help-block">

            <strong>{{ $errors->first('image') }}</strong>

        </span>

    @endif
```

```
</span>

@endif

<!-- Submit Button -->

<div class="form-group">

<button type="submit"
        class="btn btn-primary btn-lg">

    Update

</button>

</div>

</div>

</form>

@endsection

@section('scripts')

<script>

    function ConfirmDelete()
    {

        var x = confirm("Are you sure you want to delete?");
        return x;
    }

</script>

@endsection
```

Please use the Gist for code formatting.

There really isn't anything to this form we haven't seen before. You'll note that we are not including form inputs for image_name and image_extension. We are not allowing those to be modified because it makes

managing the image files too complicated for the scope of this book. If the user wants to change username, they have to delete and create a new image.

If you object to the following:

```
 {{ $marketingImage->image_name  
 . ' ' .  
 $marketingImage->image_extension }}
```

You can make a helper method, maybe something like `showImageName()`, on the model on your own to clean that up, like we do with the `show image` method.

Note if you want to change an image name, you have to delete the record and upload again. You can, on your own, make the code accommodate changing the name, however it's beyond the scope of this chapter, and in practice, I don't find this is a necessary feature for the things I'm working on. At the enterprise level, it may be necessity, but you can deal with it when you have those requirements. In most cases, like I said, I don't need it at all.

Edit Method

Ok, back to the controller, we have the edit method:

```
public function edit($id)  
{  
    $marketingImage = MarketingImage::findOrFail($id);  
  
    $thumbnailPath = $this->thumbnailPath;  
  
    return view('marketing-image.edit', compact(  
        'marketingImage', 'thumbnailPath'  
    ));  
}
```

We're just returning the view form with an instance of the model, so nothing new there.

Assuming you have a record saved, you can call the edit form via:

`sample-project.com/marketing-image/1/edit`

And obviously, you would replace 1 with actual the number of your record. Then it should something like this:

The screenshot shows a Laravel application's 'Edit Image' form. At the top, there is a dark header bar with the text 'Sample Project' on the left and navigation links 'Home', 'About', 'Users', 'Content', and 'gumby' on the right. Below the header, a breadcrumb trail shows the current path: 'Home / Marketing Images / laravel / Edit'. The main content area has a title 'Update Image'. It contains several input fields: 'Image Name' with the value 'laravel.png', a 'Delete' button, 'Thumbnail' (with a small thumbnail image), 'Is Active' (set to 'Yes'), 'Is Featured' (set to 'No'), and a 'Primary Image' field with a 'Choose File' button and a note 'No file chosen'. A large blue 'Update' button is at the bottom. At the very bottom of the page, a footer bar displays the copyright notice '© 2015 - 2017 Sample Project All rights Reserved.'

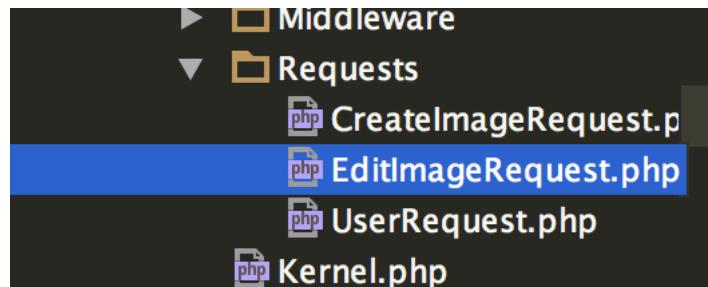
This is a just a very basic page and a starting point. Feel free to use your html and css ninja skills to make it prettier if you wish.

Edit Image Request

We are going to create a separate request class for updating. So from the command line, run the following:

```
php artisan make:request EditImageRequest
```

That will create the stub here:



Next we need to replace the stub with the following:

Gist:

[EditImageRequest](#)

From book:

```
<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class EditImageRequest extends FormRequest
{

    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */

    public function authorize()
    {

        return true;
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
}
```

```
*/  
  
public function rules()  
{  
  
    return [  
  
        'is_active' => 'required|boolean',  
        'is_featured' => 'required|boolean',  
        'image' => 'mimes:jpeg,jpg,bmp,png|max:1000'  
    ];  
  
}  
}
```

So in this case, the image is not required. We might just want to change the `is_featured` and `is_active` attributes on the model, so we don't need a file in every scenario.

Because we have this option, whereas on creating an image we do not, we have created a separate request class to handle it. It's super clean and super easy to follow.

Update method

Now we can move on to our update method on the `MarketingImageController`, which has a few more parts to it:

```
public function update($id, EditImageRequest $request)  
{  
    $marketingImage = MarketingImage::findOrFail($id);  
  
    $this->setUpdatedModelValues($request, $marketingImage);  
  
    // if file, we have additional requirements before saving  
  
    if ($this->newFileIsUploaded()) {  
  
        $this->deleteExistingImages($marketingImage);  
    }  
}
```

```
$this->setNewFileExtension($request, $marketingImage);

}

$marketingImage->save();

// check for file, if new file, overwrite existing file

if ($this->newFileIsUploaded()){

    $file = $this->getUploadedFile();
    $this->saveImageFiles($file, $marketingImage);

}

$thumbnailPath = $this->thumbnailPath;

$imagePath = $this->imagePath;

alert()->success('Congrats!', 'image edited!');

return view('marketing-image.show', compact(
    'marketingImage',
    'thumbnailPath',
    'imagePath'
));

}
```

You can see we are validating with the correct request object:

```
public function update($id, EditImageRequest $request)
{
```

Next we use findOrFail to return the model instance we want from the \$id that was handed in:

```
$marketingImage = MarketingImage::findOrFail($id);
```

Then we set the model values, using a controller method:

```
$this->setUpdatedModelValues($request, $marketingImage);
```

That method looks like this:

```
private function setUpdatedModelValues
    (EditImageRequest $request, $marketingImage)
{
    $marketingImage->is_active = $request->get('is_active');
    $marketingImage->is_featured = $request->get('is_featured');

}
```

Then if we have a new file, we delete the existing images and set the file extension because we can't assume the new file extension will be the same as the old file extension.

So we are utilizing 3 methods from the ManagesImages trait:

Here is the first one:

```
private function newFileIsUploaded()
{
    return !empty(Input::file('image'));
}
```

We just check to see if the form input for file is not empty.

Next, we delete existing images because if we overwrite with a new file that has a different file extension, it will not delete the old file, and you will be stuck with an image file that you do not need.

```
private function deleteExistingImages($modelImage)
{
    // delete old images before saving new

    $this->deleteImage($modelImage, $this->destinationFolder);

    $this->deleteThumbnail($modelImage, $this->destinationThumbnail);

}
```

You can see this method just calls two other methods, one for image and one for thumbnail:

```
Private function deleteImage($modelImage, $destination)
{
    File::delete(public_path($destination) .
        $modelImage->image_name . '.' .
        $modelImage->image_extension);

}
```

And the one for thumbnail:

```
Private function deleteThumbnail($modelImage, $destination)
{
    File::delete(public_path($destination) . $this->thumbPrefix
        . $modelImage->image_name
        . '.'
        . $modelImage->image_extension);

}
```

In both methods, we use the handy File::delete method that Laravel provides us with. Feel free to refactor those into one method if you like.

The next step back in the controller in the if block, is to assign the new image_extension attribute from the request value.

We do that using the controller method, setNewFileExtension method:

```
private function setNewFileExtension(EditImageRequest $request, $marketingImage)
{
    $marketingImage->image_extension = $request->file('image')
        ->getClientOriginalExtension();
}
```

Since this is dealing with the request that is probably specific to this marketing image type, I opted to keep this method in the controller.

Next we have our if statement to check and see if there is a file, and handle it accordingly:

```
if ($this->newFileIsUploaded()){

    $file = $this->getUploadedFile();

    $this->saveImageFiles($file, $marketingImage);

}
```

We saw all of this in the create method, so no need to get granular on it again.

Next we assign some properties to pass along to the view:

```
$thumbnailPath = $this->thumbnailPath;

$imagePath = $this->imagePath;
```

We finish up the method by alerting and returning the show view:

```
alert()->success('Congrats!', 'image edited!');

return view('marketing-image.show', compact(
    'marketingImage',
    'thumbnailPath',
    'imagePath'
));
```

Destroy Method

Let's take a look at our destroy method:

```
public function destroy($id)
{
    $marketingImage = MarketingImage::findOrFail($id);

    $this->deleteExistingImages($marketingImage);

    MarketingImage::destroy($id);

    alert()->error('Notice', 'image deleted!');

    return redirect()->route('marketing-image.index');

}
```

We use `findOrFail` to return the model, then use the model to identify which file we wish to delete. This is the second use of `deleteExistingImages`, so we have already covered that. We follow that with `destroy` method on the model to get rid of the DB record. Then we alert and return to all records.

Index method

On the controller, the index method grabs the thumbnail path and sends it along to the view:

```
public function index()
{
    $thumbnailPath = $this->thumbnailPath;

    $marketingImages = MarketingImage::paginate(10);

    return view('marketing-image.index', compact(
        'marketingImages',
        'thumbnailPath'
));

}
```

We also are paginating like we have on other index pages.

index view

Gist:

[index view](#)

From book:

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

@section('title')
    <title>Marketing Images</title>

@endsection

@section('content')
    <ol class='breadcrumb'>
        <li><a href='/'>Home</a></li>
        <li class='active'>Marketing Images</li>
    </ol>

    <h2>Marketing Images</h2>

    <hr/>

    @if($marketingImages->count() > 0)

        <table class="table table-hover table-bordered table-striped">

            <thead>
                <th>Id</th>
                <th>Thumbnail</th>
                <th>Name</th>
                <th>Date Created</th>
            </thead>

            <tbody>

                @foreach($marketingImages as $marketingImage)

```

```
<tr>

<td><a href="/marketing-image/{{ $marketingImage->id }}/edit">
{{ $marketingImage->id }}
</a>

</td>

<td><a href="/marketing-image/{{ $marketingImage->id }}">
 }})
```

```
Sorry, no Marketing Images

@endif

{{ $marketingImages->links() }}



<button type="button"
    class="btn btn-lg btn-primary">

        Create New
    </button>
</a>
</div>

@endsection


```

Please use the gist for code format.

The only thing new we did here is pass along the thumbnail variable so we can call the record's thumbnail in the table.

```
<td><a href="/marketing-image/{{ $marketingImage->id }}">

    
</a>
</td>
```

If everything is correct, and you've added a few images, it should look like this:

The screenshot shows a web application interface. At the top, there is a dark header bar with the text "Sample Project" on the left and navigation links "Home", "About", "Users", "Content", and a user profile icon "gumby" on the right. Below the header, a breadcrumb navigation bar shows "Home / Marketing Images". The main content area has a title "Marketing Images". Underneath the title is a table with four rows, each representing a marketing image. The columns are labeled "Id", "Thumbnail", "Name", and "Date Created". The first row has an id of 2, thumbnail showing a screenshot of a Laravel dashboard, name "laravel", and date "01-28-2017". The second row has an id of 3, thumbnail showing a screenshot of a terminal window with code, name "Screenshot", and date "01-28-2017". The third row has an id of 4, thumbnail showing two small circular icons, name "Wonder", and date "01-28-2017". At the bottom of the table is a blue button labeled "Create New". In the footer area, there is a copyright notice: "© 2015 - 2017 Sample Project All rights Reserved."

Id	Thumbnail	Name	Date Created
2		laravel	01-28-2017
3		Screenshot	01-28-2017
4		Wonder	01-28-2017

Add Marketing Images To Nav

Let's add it to the content dropdown above the widgets

Gist:

[dropdown nav](#)

From book:

```
@if (Auth::check() && Auth::user()->isAdmin())  
  
    <li><a href="/marketing-image">Marketing Images</a></li>  
  
@endif  
  
</ul>  
  
</li>
```

Carousel

So now we want to use our marketing images. Adding a slider or carousel to an application is a fairly common feature request, so we will build a carousel that uses our marketing images.

Pages Index View

We'll start by adding two lines to our pages.index view above the jumbotron:

```
@include('pages.slider')  
  
<br><br>
```

For reference, I'll show the code above and below, so you know where to put it.

```
<div
    class="fb-like"
    data-share="true"
    data-width="450"
    data-show-faces="true">
</div>
</div>

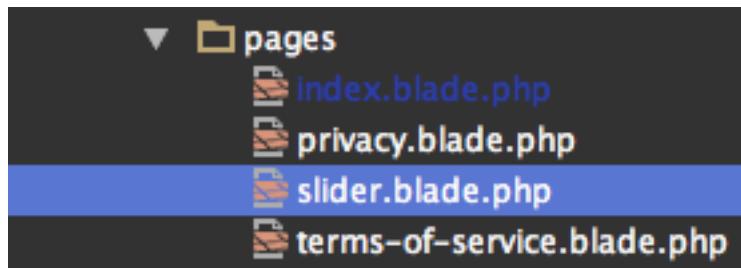
@include('pages.slider')

<br>
<br>

<!-- Main jumbotron for a primary marketing message or call to action -->

<div class="jumbotron">
```

You can see we added a couple of
 after the include just to give some margin without doing any css. Of course, we don't have the slider partial yet, so let's make that now. In your pages view folder, create a slider.blade.php file. It should look like this:



Next we will grab the carousel code from getbootstrap.com. Let's paste that into slider.blade.php:

Gist:

[carousel](#)

From book:

```
<div id="carousel-example-generic"
      class="carousel slide"
      data-ride="carousel">

  <!-- Indicators -->

  <ol class="carousel-indicators">

    <li data-target="#carousel-example-generic"
        data-slide-to="0" class="active"></li>

    <li data-target="#carousel-example-generic" data-slide-to="1"></li>

    <li data-target="#carousel-example-generic" data-slide-to="2"></li>

  </ol>

  <!-- Wrapper for slides -->

  <div class="carousel-inner" role="listbox">

    <div class="item active">

      
      <div class="carousel-caption">

        . . .

      </div>

    </div>

    <div class="item">

      
      <div class="carousel-caption">

        . . .

      </div>

    </div>

  </div>
```

```
 . . .

</div>

<!-- Controls -->

<a class="left carousel-control"
  href="#carousel-example-generic"
  role="button"
  data-slide="prev">

  <span class="glyphicon glyphicon-chevron-left"
    aria-hidden="true"></span>

  <span class="sr-only">Previous</span>

</a>
<a class="right carousel-control"
  href="#carousel-example-generic"
  role="button"
  data-slide="next">

  <span class="glyphicon glyphicon-chevron-right"
    aria-hidden="true"></span>

  <span class="sr-only">Next</span>

</a>

</div>
```

The bootstrap implementation that ships with Laravel out of the box does not include the path for glyphicons, so we will be doing our full integration next chapter when we learn about Mix.

For now, we are going to put a CDN call in place as a temporary fix. Go to your layouts/css.blade.php file and add the following as the first line:

```
<link rel="stylesheet"
      href=
"https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">

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

Obviously that should be all one line. I put the existing app.css call there for reference.
So now if you want to just check and see if your carousel is working, just list a couple of images in it like so:

```
<!-- Wrapper for slides -->

<div class="carousel-inner" role="listbox">

  <div class="item active">
    

    <div class="carousel-caption">

      <h1></h1>

    </div>
  </div>

  <div class="item">
    
    <div class="carousel-caption">

      <h1> </h1>

    </div>
  </div>

</div>
```

Obviously you would drop in your own images. Please note that if your images are of different dimensions, the size of the slider will jump around. You can solve this by using images of the same size or by writing css to scale them for you.

Our intention is to populate the slider dynamically. You can see in the above that we have an active image, which is the first one loaded in the carousel and then an image with a class of “item.”

In our `MarketingImage` model, we have `is_active` and `is_featured`. Now it’s tempting to use `is_active` to determine which image will be the active one, but we’re not going to do that.

When I think of an image as being the first one, I think of it as the featured image. So for us `is_featured` will indicate that we expect the image to be the first one.

The `is_active` attribute will be used to determine if the image is to be included in the results.

Please remember to have one and only one `is_featured` image and at least one `is_active` image in your DB if you want to see the slider function correctly.

Ok, so let’s work this up.

The first thing we are going to do is add a method to our `ShowsImages` trait, which will come in very handy for debugging purposes. Add the following method to `app/Traits/ShowsImages.php`:

Gist:

[notEnoughSliderImages method](#)

From book:

```
**  
* @param $featuredImage  
* @param $activeImages  
* @return bool  
*/  
  
private function notEnoughSliderImages($featuredImage, $activeImages)  
{  
  
    return (!count($featuredImage) ||  
        !count($activeImages) >= 1) ?  
            true : false;  
  
}
```

We’re looking for this method to return true or false, and we need at least two images for that to happen, one featured image and at least one active image.

I put this in the trait incase we want to use the method in some place other than the PagesController.
Let's look at our PagesController and add the following use statements:

```
use App\MarketingImage;
use App\Traits\ManagesImages;
use App\Traits\ShowsImages;
```

Now we'll pull in the ManagesImages trait like so:

```
class PagesController extends Controller
{
    use ManagesImages, ShowsImages;
```

Let's add a constructor:

```
public function __construct()
{
    $this->setImageDefaultsFromConfig('marketingImage');
}
```

So now we have access to our image defaults. Then let's change the index method to the following:

Gist:

[index method](#)

From book:

```
public function index()
{
    $featuredImage = MarketingImage::where('is_featured', 1)
        ->where('is_active', 1)
        ->first();

    $activeImages = MarketingImage::where('is_featured', 0)
        ->where('is_active', 1)
        ->get();

    $count = count($activeImages);

    $notEnoughImages =
        $this->notEnoughSliderImages($featuredImage, $activeImages);

    $imagePath = $this->imagePath;

    return view('pages.index', compact(
        'featuredImage',
        'activeImages',
        'count',
        'imagePath',
        'notEnoughImages'
    ));

}
```

Just a reminder to use the Gist. So you can see we are grabbing the featured image and then we have a separate query for all the other images that are active but not featured.

We also have a count variable to return the count of \$activeImages. We will use this to control the indicators for the carousel.

For reference, if you need it, here is a Gist to the full controller:

[PagesController Updated](#)

Update slider.blade.php

Gist:

slider.blade.php

From book:

```
<div>
<div>

@if ($notEnoughImages)

    Not enough images in slider.
    Populate images or remove slider include
    from pages.index.

@else

<div id="carousel-marketing-images"
      class="carousel slide"
      data-ride="carousel"
      data-interval="false">

    <!-- Indicators -->

    <ol class="carousel-indicators">

        <li data-target="#carousel-marketing-images"
            data-slide-to="0"
            class="active"></li>

    @foreach (range(1, $count) as $number)

        <li data-target="#carousel-marketing-images"
            data-slide-to="{{ $number }}></li>

    @endforeach

</ol>

<!-- Wrapper for slides -->

<div class="carousel-inner" role="listbox">
```

```
<div class="item active">

image_name }}">

<div class="carousel-caption">
    <h1></h1>
</div>
</div>

@foreach ($activeImages as $image)

<div class="item">

image_name }}">

<div class="carousel-caption">
    <h1> </h1>
</div>
</div>

@endforeach

</div>

<!-- Controls -->

<a class="left carousel-control"
   href="#carousel-marketing-images"
   role="button"
   data-slide="prev">

    <span class="glyphicon glyphicon-chevron-left"
          aria-hidden="true"></span>

    <span class="sr-only">
```

Previous

```
</span>

</a>

<a class="right carousel-control"
   href="#carousel-marketing-images"
   role="button"
   data-slide="next">

  <span class="glyphicon glyphicon-chevron-right"
        aria-hidden="true"></span>
  <span class="sr-only">

    Next

  </span>

</a>

</div>

@endif

</div>
</div>
```

Please use the Gist for code format.

Now if you look at the index page and it doesn't show any images, make sure you have images in the marketing-images table with `is_active` and `is_featured` appropriately set.

Ok, we have interesting things in here. First, we did not insert any dynamic content into the caption section. I find in practice I don't use it, so I skipped it. If you would like to use this property, you need to add it to the `MarketingImage` model and table.

So the first thing to note is that I wrapped most of the carousel in an if statement:

```
<div>
  <div>

    @if ($notEnoughImages)

      Not enough images in slider.
      Populate images or remove slider include
      from pages.index.

    @else
```

This is just for debug purposes. It will remind you if you forget to save images for the slider. If someone clones this repository from Github, at least it will run without errors.

The next bit of dynamic content that we are doing is for the indicators:

```
<!-- Indicators -->

<ol class="carousel-indicators">

  <li data-target="#carousel-marketing-images"
      data-slide-to="0"
      class="active"></li>

  @foreach(range(1, $count) as $number)

    <li data-target="#carousel-marketing-images"
        data-slide-to="{{ $number }}></li>

  @endforeach

</ol>
```

You can see we are using the PHP's range function to create a range from 1 to the count of the \$count variable, so we get the right number of indicators. Hopefully that is clear.

Our next dynamic piece is the featured image inside the item active class.

```
<!-- Wrapper for slides -->

<div class="carousel-inner"
    role="listbox">

    <div class="item active">

        image_name }}"

        <div class="carousel-caption">

            <h1></h1>

        </div>
    </div>
</div>
```

Note that we are using our showImage method and \$imagePath variable to set the image path. For the rest of the images, we use a foreach loop:

```
@foreach ($activeImages as $image)

    <div class="item">

        image_name }}"

        <div class="carousel-caption">

            <h1> </h1>

        </div>
    </div>

@endforeach
```

And that's pretty much it. We now have a dynamic carousel that will populate according to how you manage the marketing images.

To center your images, you can use some style to control them. Add the following to layouts/css.blade.php in between the style tags.

Gist:

[css for carousel](#)

From book:

```
.carousel-inner img {  
    margin: auto;  
}
```

Feel free to work with that style as you wish. Obviously adding style here is not a permanent solution, we will be refactoring it into our resources by using Mix in the next chapter.

image_weight

One feature that would be nice to hand to the client is to give them control over the order of the active images.

We already know that the featured image will come first. But what if they want to control the order of the remaining images?

We can easily accomplish this with a simple modification to our marketing_images table, along with corresponding changes to our model, MarketingImageController and views.

We will start by creating the migration from the command line:

```
php artisan make:migration add_image_weight_to_marketing_images_table
```

Now we can modify the migration file to the following:

Gist:

[migration](#)

From book:

```
<?php

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

class AddImageWeightToMarketingImagesTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */

    public function up()
    {
        Schema::table('marketing_images', function(Blueprint $table)
        {

            $table->integer('image_weight')
                ->default('100')
                ->after('image_extension');

        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */

    public function down()
    {
        Schema::table('marketing_images', function ($table) {
            $table->dropColumn('image_weight');
        });
    }
}
```

We will default to 100, which make the image a low priority, since we sort ascending by image_weight. In that case 1, will be the highest priority.

Let's run:

```
php artisan migrate
```

Next, let's add the attribute to our MarketingImage model in the \$fillable array:

```
protected $fillable = ['is_active',
                      'is_featured',
                      'image_name',
                      'image_extension',
                      'image_weight'];
```

Just one line needed, so no Gist.

Modify MarketingImageController

For reference, I will give you the entire controller in a gist.

Gist:

[MarketingImageController](#)

We only need to add two lines however. We need the following line added to the store method:

```
'image_weight' => $request->get('image_weight')
```

And we need one line added to the setUpdatedImageValues method at the bottom of the class:

```
$marketingImage->image_weight = $request->get('image_weight');
```

CreateImageRequest and EditImageRequest

We also need to add the following method to the rules in both CreateImageRequest.php and EditImageRequest.php:

```
'image_weight' => 'integer|between:1,100',
```

Marketing Image Create and edit Views

Add the following form input for the create view:

Gist:

[form input](#)

From book:

```
<!-- image_weight Form Input -->

<div class="form-group{{ $errors->has('image_weight') ? ' has-error' : '' }}>

    <label class="control-label">Image Weight</label>

    <input type="number"
        class="form-control"
        name="image_weight"
        value="{{ old('image_weight') ? old('image_weight') : 100 }}>
```

```
@if ($errors->has('image_weight'))  
  
    <span class="help-block">  
  
        <strong>{{ $errors->first('image_weight') }}</strong>  
  
    </span>  
  
@endif  
  
</div>
```

You can see we are using a ternary to supply either the old value or 100 as a default. Just a note. If you are running PHP 7, you can write that ternary as follows:

```
old('image_weight') ? : 100
```

On the edit.blade.php, we have a slight variation:

```
<!-- image_weight Form Input -->  
  
<div class="form-group{{ $errors->has('image_weight') ? ' has-error' : '' }}>  
  
    <label class="control-label">Image Weight</label>  
  
    <input type="number"  
          class="form-control"  
          name="image_weight"  
          value="{{ old('image_weight') ?  
                  old('image_weight') :  
                  $marketingImage->image_weight }}"/>
```

```
@if ($errors->has('image_weight'))  
  
    <span class="help-block">  
  
        <strong>{{ $errors->first('image_weight') }}</strong>  
  
    </span>  
  
@endif  
  
</div>
```

This will default to the image_weight on the model record or to the old method if you are submitting and you get some other error.

MarketingImage Show View

Add the following to the table in the show.blade.php view. The gist will have the full file.

Gist:

[show.blade.php](#)

From book:

```
<tr>  
  
    <th>Image Weight</th>  
  
</tr>  
  
<tr>  
  
    <td>{{ $marketingImage->image_weight }}</td>  
  
</tr>
```

PagesController Index Method

One last change to the index method on the PagesController:

```
public function index()
{
    $featuredImage = MarketingImage::where('is_featured', 1)
        ->where('is_active', 1)
        ->first();

    $activeImages = MarketingImage::where('is_featured', 0)
        ->where('is_active', 1)
        ->orderBy('image_weight', 'asc')
        ->get();

    $count = count($activeImages);

    $notEnoughImages =
        $this->notEnoughSliderImages($featuredImage, $activeImages);

    $ImagePath = $this-> imagePath;

    return view('pages.index', compact(
        'featuredImage',
        'activeImages',
        'count',
        'ImagePath',
        'notEnoughImages'
    ));
}
```

You can see we just added to the query for \$activeImages:

```
->orderBy('image_weight', 'asc')
```

grid.blade.php

I will provide the entire file in the gist, but we are only changing one block.

Gist:

[grid.blade.php](#)

From book:

```
<div class="col-xs-6 col-lg-4">

    <a href="/marketing-image"><h2>Marketing Images</h2></a>
    <p>
        <a href="/marketing-image">

            Use this link to manage your marketing images

        </a>
        </p>
        <p><a class="btn btn-default"
            href="/marketing-image"
            role="button">

            View details &raquo;

        </a>
        </p>
    </div><!--/.col-xs-6.col-lg-4-->
```

Reviewing this workflow, I noticed it would be useful to see the image weight as well as is_featured and is_active on the grid view on the index page of MarketingImage. So let's make that happen. We will step through it quickly:

MarketingImageController Index Method

```
public function index()
{
    $thumbnailPath = $this->thumbnailPath;

    $marketingImages = MarketingImage::orderBy('image_weight', 'asc')
        ->paginate(10);

    return view('marketing-image.index', compact(
        'marketingImages',
        'thumbnailPath'
    ));
}
```

You can see paginate comes last in query, which makes a lot of sense.

marketing-image index.blade.php

Gist:

[table for index.blade.php](#)

From book:

```
<table class="table table-hover table-bordered table-striped">

<thead>

<th>Thumbnail</th>
<th>Name</th>
<th>Weight</th>
<th>Featured</th>
<th>Active</th>
<th>Date Created</th>

</thead>
```

```
<tbody>

@foreach($marketingImages as $marketingImage)

<tr>

<td><a href="/marketing-image/{{ $marketingImage->id }}">

</a></td>

<td><a href="/marketing-image/{{ $marketingImage->id }}">
{{ $marketingImage->image_name }}</a></td>

<td>{{ $marketingImage->image_weight }}</td>

<td>{{ $marketingImage->showFeaturedStatus($marketingImage->is_featured) }}</td>

<td>{{ $marketingImage->showActiveStatus($marketingImage->is_active)}}</td>

<td>{{ $marketingImage->created_at }}</td>

</tr>

@endforeach

</tbody>

</table>
```

Please use the Gist for code format.

It should look like this when you are done:

The screenshot shows a web application interface titled "Sample Project". At the top, there is a navigation bar with links for "Home", "About", "Users", "Content", and a user profile for "gumby". Below the navigation bar, the page title is "Marketing Images". The main content area displays a table with three rows of data. The columns are labeled "Thumbnail", "Name", "Weight", "Featured", "Active", and "Date Created". The data is as follows:

Thumbnail	Name	Weight	Featured	Active	Date Created
	laravel	90	No	Yes	01-28-2017
	Screenshot	100	No	Yes	01-28-2017
	Wonder	100	Yes	Yes	01-28-2017

Below the table, there is a blue button labeled "Create New". At the bottom of the page, a copyright notice reads "© 2015 - 2017 Sample Project All rights Reserved."

You can see we are ordered by the lowest weight number. I got rid of the ids because we don't need them. So wouldn't it be great if we could do column sorts?

That is exactly the problem we are going to solve by using Vue.js.

Summary

We got granular with image management in this chapter to get you familiar with how we manage files and how they relate to models. You can see it's a loose association in some ways and we have to be careful to make sure we are coordinating actions accordingly.

We also built a dynamically populated bootstrap carousel. We are working with bootstrap because it gives us a consistent theme that is fairly easy to replace if we choose to do so.

There are many different jquery sliders out there, so feel free to pick the one of your choosing and modify the code as you see fit. This chapter should give a good idea on how to approach that.

Chapter 11 Introducing Mix and Vue.js

One of the most amazing components to the Laravel framework is Mix, which provides a fluent interface for compiling assets such as javascript and css.

So you might ask, why do I need this? Don't I already have an css/app.css and js/app.js file in the public folder? Yes you do, and you do not have to use Mix, it is not required. But Mix opens you up to a whole new world of development and you are going to want to use it. It's awesome.

When I've written about Laravel in the past, I always avoided this topic because I thought it was too advanced, and set up was always a little difficult.

With Laravel 5.4, things are significantly easier, it ships with defaults that make setup very easy.

The [docs are here](#) and are easy and clear to follow.

Node

You will of course have to have node setup. You can test that by running the following from the command line:

```
node -v
```

That should return the version number you are running of node. As of this writing, I'm running 7.4.0. I'm not sure what the minimum version of node is for Laravel Mix, but I do know I had to update to get it to work. The Laravel docs do not list the minimum requirement, so to save yourself a headache, I would just use 7.4.0 as a minimum version as of this writing.

If you need to install node, you can do so from here:

[node.js](#)

To repeat, since Mix is brand new with Laravel 5.4, replacing Elixir, I'll assume you need the latest stable version of node. If you are running a version less than that, you may experience problems.

Also note, users have had issues installing node properly on windows, so for those users, I'm including a quote from the following article:

<https://laravelista.com/lessons/sublimely-magnificent-laravel-mix>

"In the Laravel Mix documentation it says that if you are running a VM on a Windows host system, you may need to run npm install --no-bin-links. Don't do it!"

If you have Node.js and NPM installed on Windows, and are using Git Bash (which comes with Git; with symbolic links enabled) then everything will work correctly. Using --no-bin-links just makes a huge mess on the disk and takes much more disk space."

Ok, moving on. There's a really cool version manager for node name "n," which is the first time I've come across a package with a single letter for a name. How cool is that?

It's not really a beginner topic, but I'm going to include this link to [n](#) in case you want to check it out.

NPM

You also need to test for npm, the package manager with this command from the command line:

```
npm -v
```

As of this writing, I'm running 3.8.3.

If you need to install node, you can do so from here:

```
npm
```

Running NPM Install

Ok, now that we have that out of the way, let's install Laravel Mix itself. We do this with a simple command:

```
npm install
```

If you are on a windows machine, you may have to do the following:

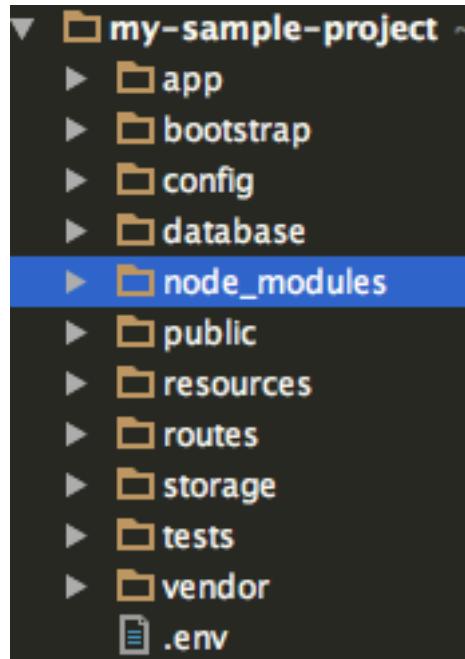
```
npm install --no-bin-links
```

Upon successful install, you will get the following:

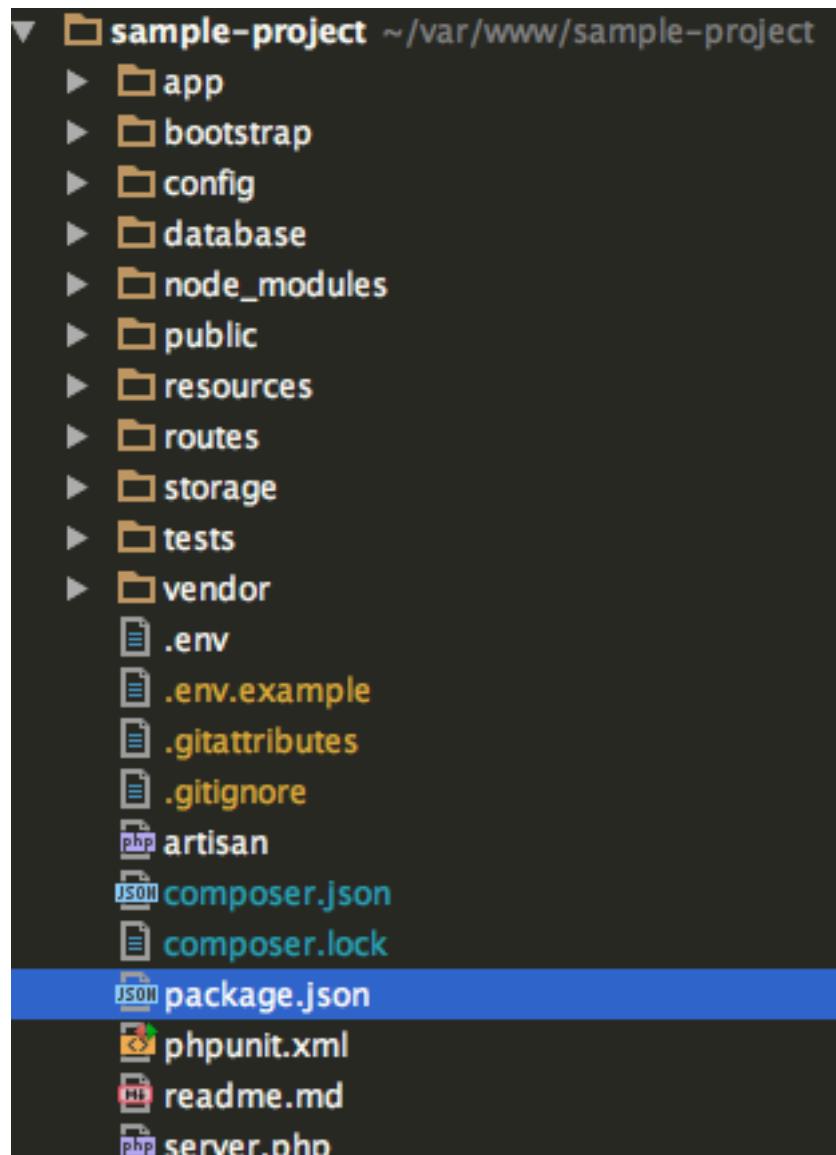
```
obuf@1.1.1
└─ wbuf@1.7.2
    └── minimalistic-assert@1.0.0
├─ strip-ansi@3.0.1
├─ ansi-regex@2.1.1
└─ supports-color@3.2.3
├─ webpack-dev-middleware@1.9.0
└─ yargs@6.6.0
    ├─ camelcase@3.0.0
    └─ yargs-parser@4.2.1
├─ webpack-md5-hash@0.0.5
└─ md5@2.2.1
    ├─ charenc@0.0.2
    ├─ crypt@0.0.2
    └─ is-buffer@1.1.4
├─ webpack-notifier@1.5.0
└─ node-notifier@4.6.1
    ├─ cli-usage@0.1.4
    └─ marked@0.3.6
        └─ marked-terminal@1.7.0
            ├─ cardinal@1.0.0
            │ ├─ ansicolors@0.2.1
            │ └─ redeyed@1.0.1
            └─ esprima@3.0.0
        ├─ cli-table@0.3.1
        └─ colors@1.0.3
    └─ node-emoji@1.5.1
        └─ string.prototype.codepointat@0.2.0
└─ growly@1.3.0
└─ lodash.clonedeep@3.0.2
    ├─ lodash._baseclone@3.3.0
    │ ├─ lodash._arraycopy@3.0.0
    │ ├─ lodash._arrayeach@3.0.0
    │ └─ lodash._basefor@3.0.3
    └─ lodash._bindcallback@3.0.1
└─ minimist@1.2.0
└─ shellwords@0.1.0
└─ webpack-stats-plugin@0.1.4
└─ lodash@4.17.4
└─ vue@2.1.10
```

I only provided a screenshot of the partial list.

So now in your project folder, you will see there is new folder, which holds all your node modules:



And if you open that, you can see there are lot of folders. We don't need to review them. But you also might want to take a look at your package.json file:



As of this writing, that contains the following:

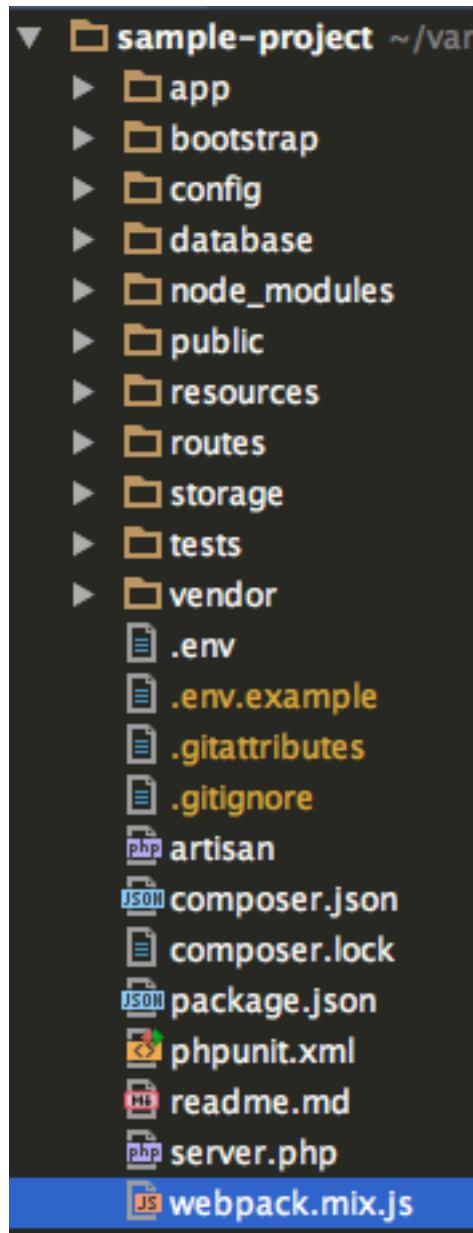
```
{  
  "private": true,  
  "scripts": {  
    "dev": "node_modules... ",  
    "watch": "node_modules... ",  
    "hot": "node_modules... ",  
    "production": "node_modules... "  
  },  
  "devDependencies": {  
    "axios": "^0.15.2",  
    "bootstrap-sass": "^3.3.7",  
    "jquery": "^3.1.0",  
    "laravel-mix": "^0.5.0",  
    "lodash": "^4.16.2",  
    "vue": "^2.0.1"  
  }  
}
```

Note: I chopped out the long scripts paths, but you can refer to them in your IDE.

In the dependencies, we are getting axios, bootstrap sass, jquery, laravel-mix, lodash, and vue. Since we've been working with bootstrap, we will assume you know what that is. Sass is an extension to css, a preprocessor that allows developers to take advantage of advanced scripting capabilities, and then compile it down to css.

Here is an [excellent article](#) on Sass.

Don't be worried about compiling it, Mix compiles if for us automatically when we run dev from the command line.



Here's what you get out of the box:

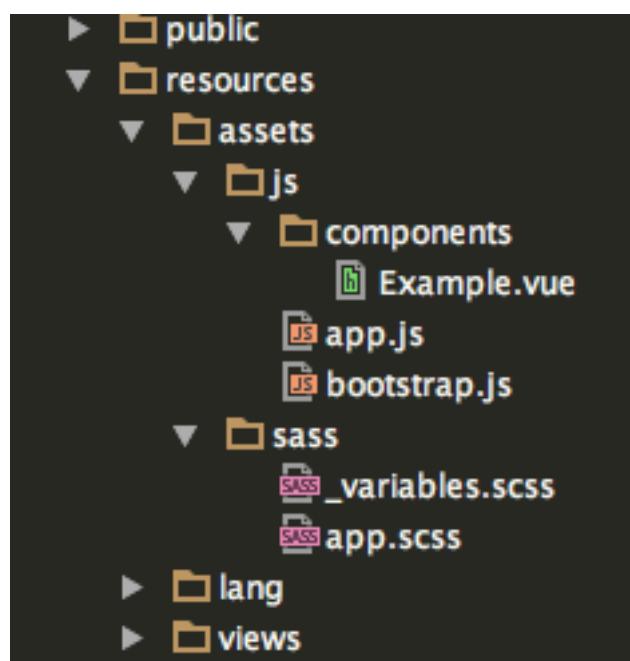
```
const { mix } = require('laravel-mix');

/*
-----
/ Mix Asset Management
-----
/
/ Mix provides a clean, fluent API for defining some Webpack build steps
/ for your Laravel application. By default, we are compiling the Sass
/ file for the application as well as bundling up all the JS files.
/
*/
mix.js('resources/assets/js/app.js', 'public/js')
    .sass('resources/assets/sass/app.scss', 'public/css');
```

For most browsers, you can't just require another javascript file like so:

```
const { mix } = require('laravel-mix');
```

But Mix, by utilizing Webpack, makes this possible because Mix uses babel to translate ES6, which is the latest version of javascript that is not fully supported in all browsers, and compiles it down to a javascript file that most browsers understand. We will see that in action in a moment. First, let's look at some of the folders we have in the resources/assets folder:



So within the assets/js folder, we have app.js, bootstrap.js, and in the components folder, we have Example.vue, which is a Vue.js component. One of the really cool things here is that Laravel sets up an example for Vue.js for us to follow, which makes the architecture behind Vue much easier to understand. You'll be amazed at how fast we get up and running with it.

For those of you unfamiliar with [Vue.js](#), it is a javascript framework that is popular with Laravel developers. We will be looking at the Vue component example in a couple of minutes.

For now, let's go back to our webpack.mix.js.

So what's happening here is that when you run npm run dev from the command line, it executes the following:

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

So this will take your app.scss, which is in your sass file and transpile it to your public/css/app.css file. The mix.sass method allows you to compile app.scss into CSS.

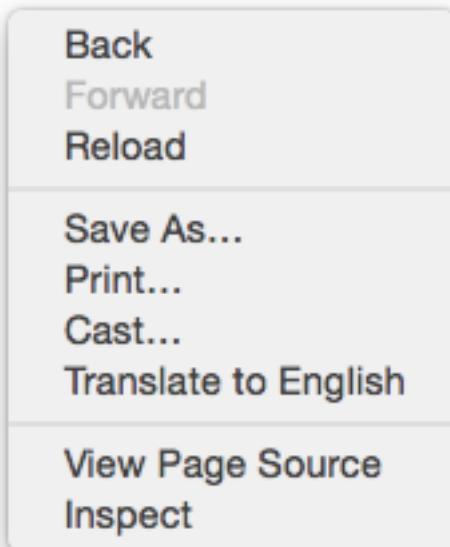
If you try that and get an error, please check your node installation, especially if you are on Windows. You can refer back to the beginning of the chapter.

One other tip, this could apply to everyone. Some users have reported needing the following in scripts.blade:

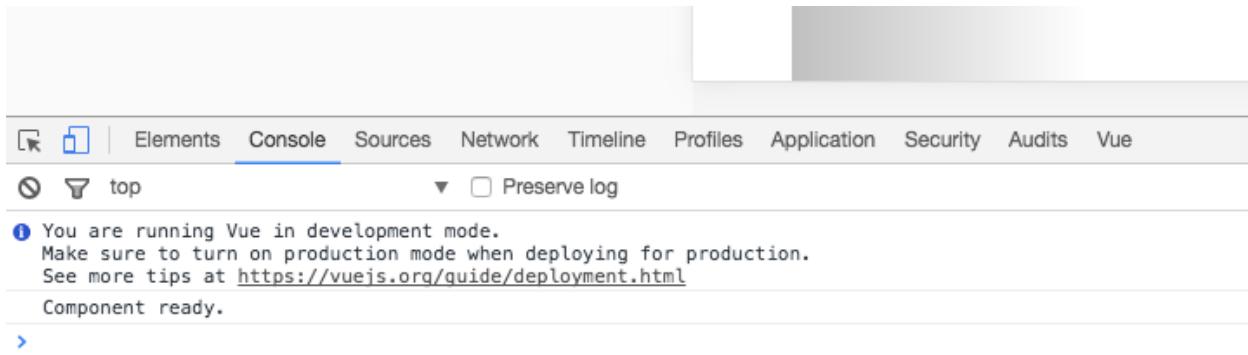
```
<script>
    window.Laravel = {!! json_encode([
        'csrfToken' => csrf_token(),
    ]) !!};
</script>
```

In my last two installs, I did not need that token there, so I'm just including that as a trouble-shooting tip, incase you need it.

Also, another trouble shooting tip. Right click on your chrome browser somewhere on the body of the page, and you will get a menu that includes inspect:



Select inspect, which will split the screen and default to the elements tab. Select the console tab:



You might not see the info on Vue in your console window, but if not, you will later on. The console window is where you will get feedback on javascript and asset errors, and it's very helpful for troubleshooting problems. Just something to keep in mind if you get stuck.

Ok, back to the good news, which is that you can take full advantage of Sass's advanced scripting capabilities. Personally, I haven't used Sass much, but a lot of front end developers love it. If you are new to Sass, you might want to check out this [quick-start tutorial on Sass](#).

Compiling Multiple Assets

If you want to compile multiple sass files, the best way to do this is to use the @import directive. If you check resources/assets/sass/app.css, you can see it's using:

```
// Variables
@import "variables";
```

This brings in some default Laravel styling. You can see by the way, that it's referencing _variables, and the underscore is a sass convention that is not needed when you are using @import, so in this case it knows that @import "variables"; is referring to "_variables."

Later, when we create a main.scss file, we will use the @import directive to pull it in. But for now, we will start more simply.

Let's make a few simple changes to our resources/layouts/css.blade.php file and try this. Lets first cut out the call to the bootstrap CDN, since we are going to fix the font problem in a minute, and this call will no longer be necessary.

Next, let's copy/cut out the css code that we have in our layouts/css.blade.php file:

```
body{
    padding-top: 65px;
    background-color: white;
}

.circ{
    width : 40px;
    vertical-align: middle;
    border-radius: 50%;
    box-shadow: 0 0 0 2px #fff;
    margin-top: 5px;
}

.circ:hover{
    box-shadow: 0 0 0 3px #f00;
```

```
}

.carousel-inner img {

margin: auto;

}
```

Notice we got rid of the <style></style> tags, we no longer need them. Let's simply paste the above code into your resources/assets/sass/app.scss file, so the whole file looks like this:

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

// Variables
@import "variables";

// Bootstrap
@import "node_modules/bootstrap-sass/assets/stylesheets/bootstrap";

body{
  padding-top: 65px;
  background-color: white;
}

.circ{

width : 40px;
vertical-align: middle;
border-radius: 50%;
box-shadow: 0 0 0 2px #fff;
margin-top: 5px;

}

.circ:hover{

box-shadow: 0 0 0 3px #f00;
```

```
}
```

```
.carousel-inner img {
```

```
margin: auto;
```

```
}
```

When you have a lot of css, you can create a separate file, but since we don't have much, we put it all in one file.

Next, we need to access the glyphicons, so we are going to add a .copy method in our webpack.mix.js by chaining it fluently after the webpack method like so:

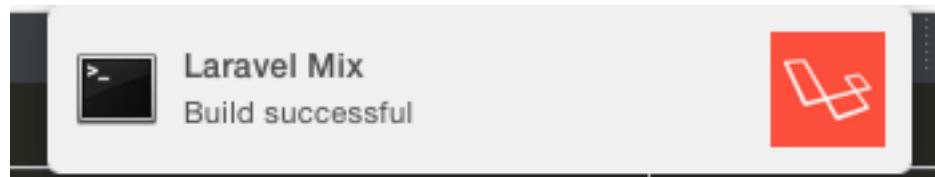
```
mix.js('resources/assets/js/app.js', 'public/js')
.sass('resources/assets/sass/app.scss', 'public/css')
.copy('node_modules/bootstrap-sass/assets/fonts/bootstrap/',
'public/fonts/bootstrap');
```

We asking Mix to copy the files in the node_modules/bootstrap-sass/assets/fonts/bootstrap folder and place them in a fonts/bootstrap folder inside of our public folder, because that is how the fonts are referenced in our compiled css file.

So let's try this and see what we get. Run the following from your command line:

```
npm run dev
```

You will get a confirmation message in the upper right of your monitor screen:



If you got to that point, that means you were successful with your css and js compilation. You may have to do a hard refresh on the page to get it to work right, since we are not using asset cache busting yet. We will do that in a minute.

A quick note about working with Mix. Babel is very unforgiving. If you have a comma out of place in your javascript, it will not compile your files, so be very careful when working with it.

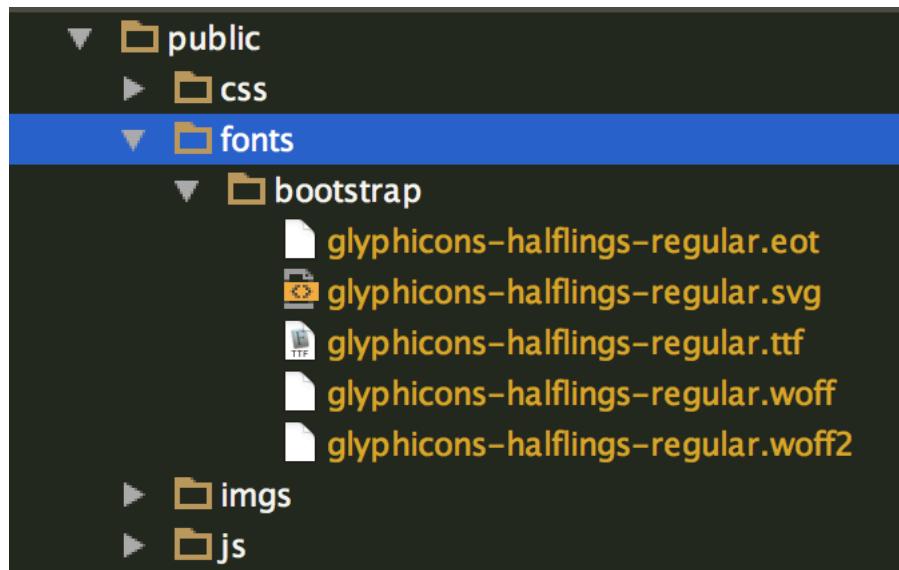
Also note, the other npm commands for Mix:

```
npm run watch // runs mix automatically, whenever you change a file
npm run production // runs mix and minifies
npm run hot // watches and refreshes the browser
```

You can end npm run watch by control c.

Ok, now if you test everything you had before, the slider should work and you should be able to see the left right glyphicon arrows as before.

You'll note that in your public folder, you now have:



How cool is that? Mix copied the files for us perfectly.

Versioning

Another super-cool feature of Mix is versioning. Sometimes you need to force browsers to load a fresh asset instead of the cached copies.

Mix has a simple version method that takes care of this for us, if we just chain the following onto the method:

```
.version();
```

So let's make the following change to our webpack.mix.js file:

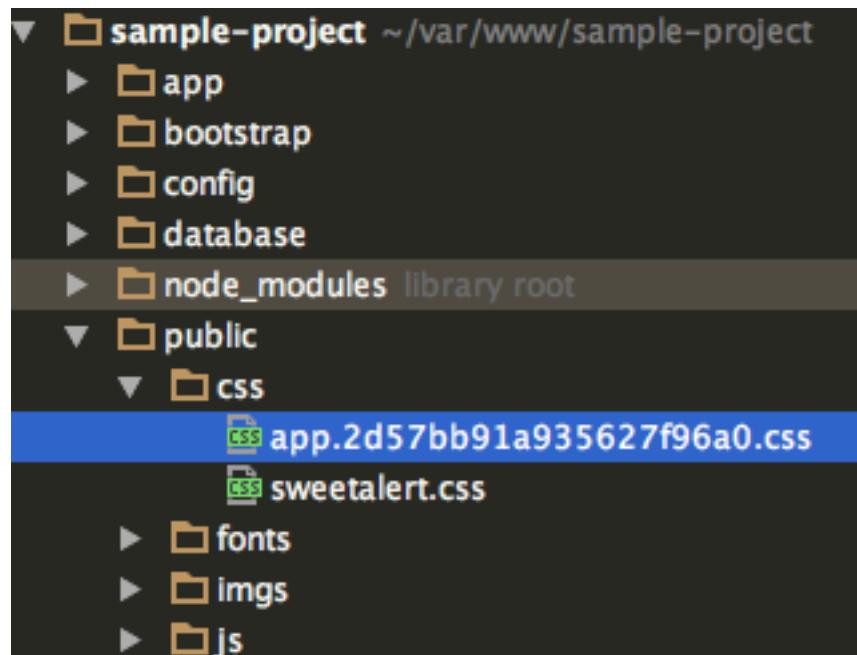
```
mix.js('resources/assets/js/app.js', 'public/js')
    .sass('resources/assets/sass/app.scss', 'public/css')
    .version();
```

Note that we chopped out the copy method. We don't need it, since we don't need to copy the fonts more than once.

So let's compile that by running:

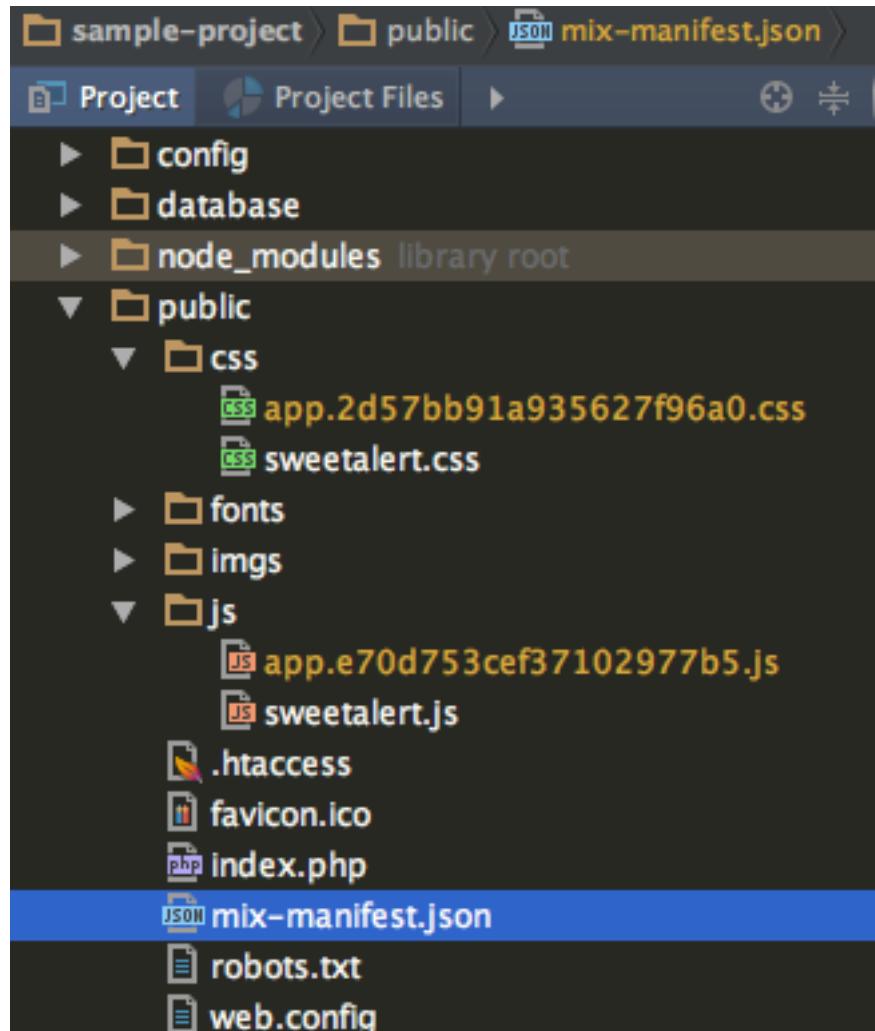
```
npm run dev
```

If you look in your public/css folder, you will see the /css/app.css file has a number appended to it:



You will find the same kind of extension on your app.js file.

You will also get a mix-manifest.json file, which aliases the versions of the css and js files.



The contents of that file: _____

```
{  
  "/js/app.js": "/js/app.e70d753cef37102977b5.js",  
  "/css/app.css": "/css/app.2d57bb91a935627f96a0.css"  
}
```

In order for this to work correctly in the browser, we need the mix helper in our resources/views/layouts/css File, so let's change the call to our css to:

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

That automatically pulls in the latest version, so you don't have to worry about naming the file.

We need to make the following change in our resources/views/layouts/scripts.blade.php file:

```
<script src="{{ mix('/js/app.js') }}"></script>
```

And now congrats, you have versioning implemented with Mix. When you finally go to production, you'll want to run:

```
npm run production
```

That will minify the files.

Vue.js

We touched on vue.js briefly at the beginning of the chapter. Laravel comes with an example component of vue setup for us, so to check that out, we need to place `<example></example>` within resources/views/-pages/index.blade.php like so:

```
<example></example>  
@include('pages.slider')
```

The example tags are a call to the Vue.js component. I left the include for the slider in for reference.

One more thing we need to do is add a div into the master page, which will define the area that Vue will be applied to:

```
<!DOCTYPE html>
<html lang="en">
<head>

    @include('layouts.meta')

    @yield('title')

    @include('layouts.css')

    @yield('css')

</head>

<body role="document">

    @include('layouts.facebook')

    <div id="app">

        @include('layouts.nav')

        <div class="container theme-showcase" role="main">

            @yield('content')

            @include('layouts.bottom')

        </div>

    </div>

    @include('layouts.scripts')

    @include('Alerts::show')

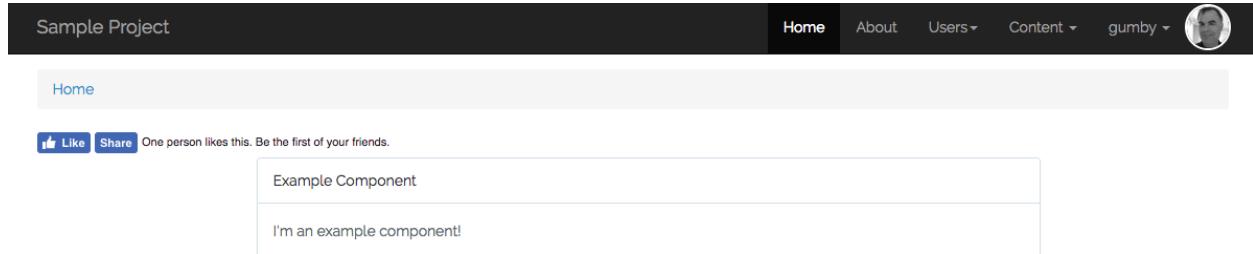
    @yield('scripts')

</body>
</html>
```

You can see the div we added has an id of ‘app.’ It opens below the body tag and closes above the include for

scripts. Note that the facebook include is outside of the div because if you run other scripts within your Vue instance, you will get a warning or possibly an error, so be sure to watch out for that.

With that in place, if you load the index page now, you will see:

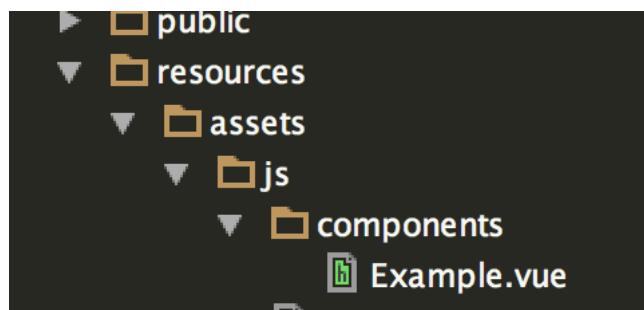


One thing I love about Vue is how modular it is. We have hardly any markup in the main view file:

```
<example></example>
```

So the question is, how do we get the result on the page? Ok, let's follow the trail.

Take a look at resources/assets/js/components/Example.vue. You can find it here:



Here are the contents of the file:

```
<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>
```

```
        </div>
    </div>
</div>
</div>
</template>

<script>

export default {

mounted() {

    console.log('Component ready.')
}

}

</script>
```

So you can see we have html markup inside of `<template></template>` tags. This is followed by the component script:

```
<script>

export default {

mounted() {

    console.log('Component ready.')
}

}

</script>
```

The mounted method in vue is a life-cycle hook that executes after the instance has been mounted to the DOM , and in this case, it is logging to the console a message to let us know the component is ready. It will also load the template in the file, which is what builds the output we eventually see on the page. You don't have to explicitly tell the script to use the template, so this is very efficient syntax.

Having the component and the html for the component extracted out to the .vue file makes for great code separation.

Now if you simply placed the script in the scripts.blade.php file, it wouldn't work, and that's because a Vue component has to be consumed by a root instance of Vue. And for that we need to look at resources/assets/js/app.js:

```
/**  
 * First we will load all of this project's JavaScript dependencies which  
 * include Vue and Vue Resource. This gives a great starting point for  
 * building robust, powerful web applications using Vue and Laravel.  
 */  
  
require('./bootstrap');  
  
/**  
 * Next, we will create a fresh Vue application instance and attach it to  
 * the body of the page. From here, you may begin adding components to  
 * the application, or feel free to tweak this setup for your needs.  
 */  
  
Vue.component('example', require('./components/Example.vue'));  
  
const app = new Vue({  
    el: '#app'  
});
```

So the first thing is that we pull in the bootstrap file in the same folder:

```
require('./bootstrap');
```

Let's take a look at that:

```
window._ = require('lodash');

/**
 * We'll load jQuery and the Bootstrap jQuery plugin which provides support
 * for JavaScript based Bootstrap features such as modals and tabs. This
 * code may be modified to fit the specific needs of your application.
 */

window.$ = window.jQuery = require('jquery');

require('bootstrap-sass');

/**
 * Vue is a modern JavaScript library for building interactive web interfaces
 * using reactive data binding and reusable components. Vue's API is clean
 * and simple, leaving you to focus on building your next great project.
 */

window.Vue = require('vue');

/**
 * We'll load the axios HTTP library which allows us to easily issue requests
 * to our Laravel back-end. This library automatically handles sending the
 * CSRF token as a header based on the value of the "XSRF" token cookie.
 */

window.axios = require('axios');

window.axios.defaults.headers.common = {
  'X-CSRF-TOKEN': window.Laravel.csrfToken,
  'X-Requested-With': 'XMLHttpRequest'
};

/**
 * Echo exposes an expressive API for subscribing to channels and listening
 * for events that are broadcast by Laravel. Echo and event broadcasting
 * allows your team to easily build robust real-time web applications.
 */

// import Echo from "laravel-echo"

// window.Echo = new Echo({
//   broadcaster: 'pusher',
//   key: 'your-pusher-key'
```

```
// });
```

You can see it pulls in the lodash js library, which is a toolkit for managing objects and collections.

Next we pull in jQuery and the Bootstrap jQuery plugin:

```
window.$ = window.jQuery = require('jquery');

require('bootstrap-sass');
```

Then we bring in Vue:

```
window.Vue = require('vue');
require('vue-resource');
```

Next we call the axios library, which handles ajax calls. We also have some code to place to set the header of outgoing ajax requests to include the CSRF token:

```
window.axios = require('axios');

window.axios.defaults.headers.common = {
  'X-CSRF-TOKEN': window.Laravel.csrfToken,
  'X-Requested-With': 'XMLHttpRequest'
};
```

Then there is commented code for Laravel's Echo configuration, which helps you integrate pusher, which we are covering later in this book, but don't need to worry about it for now.

So this bootstrap file sets some handy defaults for us that the app.js file calls. So let's go back to app.js and pick up where we left off:

```
Vue.component('example', require('./components/Example.vue'));  
  
const app = new Vue({  
  
  el: '#app'  
  
});
```

The first line pulls in the component:

```
Vue.component('example', require('./components/Example.vue'));
```

That's all you have to do to allow your main instance access to the component. We give the component its name and supply the path, and that's it. This is super simple.

This is followed by the root instance of vue:

```
const app = new Vue({  
  
  el: '#app'  
  
});
```

All we are doing here is binding the instance of Vue to the div with an id of 'app,' so el stands for element. So in this case, when the page is called, anything within the div 'app' will have access to Vue components.

Then finally, in our pages/index.blade.php view file, we added the following to call the component:

```
<example></example>
```

Personally I find this to be a very accessible implementation of a very powerful javascript framework.

Vue Basics

In order to understand how we are going to use Vue components to build our datagrids and charts, we need to know a little bit more about vue.

By no means am I an expert in Vue or javascript, so this is in no way a substitute for learning more about Vue on your own. On that note, Laracasts has a great series on Vue:

[Laracasts Vue.js](#)

Also here is the official Vue.js site:

[Vue.js](#)

I read the following book and highly recommend it:

[The Majesty of Vue 2](#)

There's also a google Chrome Vue.js extension for debugging:

[Vue Dev Tools](#)

My instructions on Vue.js are going to lean heavily in the direction of how we're actually going to use Vue. It's more practical usage than comprehensive instruction.

So on that note, let's play around with our <example> component to demonstrate some of the features of Vue.

We'll start by demonstrating Vue's two-way model binding. Let's change resources/assets/js/components/Example.vue to the following:

Gist:

[Example.vue](#)

From book:

```
<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 id="name-form">
            <label for="name">Enter name:</label>
            <input type="text" v-model="name" id="name" name="name" />
            <p>{{ name }}</p>
          </div>
        </div>
      </div>
```

```
</div>
</div>
</div>
</template>

<script>

export default {

  data : function(){

    return {

      name: ''


    }

  },


  mounted() {

    console.log('Component ready.')

  }

}

</script>
```

Ok, so the first change is in the template:

```
<div id="name-form">

<label for="name">Enter name:</label>

<input type="text" v-model="name" id="name" name="name" />

<p>{{ name }}</p>

</div>
```

You can see it's a typical input form, except for:

```
v-model ="name"
```

We are using the v-model directive to bind the name input to the name data on the vue model in the component.

Below the form, we have:

```
<p>{{ name }}</p>
```

The double-braces in this case are for Vue, and they will return the value of name. Now the double-brace syntax is the same as blade, but because this is not a blade file, there is no conflict and we are getting our output from Vue. If you want to use Vue this way directly in a blade file, you would have to escape it with an @ sign in front like:

```
@{{ data }}
```

That way blade will not compile it.

But we have no need for that, since we are not embedding directly in our blade view files, we are using components instead, which gives us a high-degree of code separation re-usability.

Ok, next we need to look at the component itself:

```
<script>

  export default {

    data : function(){

      return {

        name: ''



      }
    }

  },
```

```
mounted() {  
  
    console.log('Component ready.')  
  
}  
  
}  
  
</script>
```

You can see we added a data object that returns an empty value for name. So this is what we get when we load the page. Because we are binding the input via our v-model directive to our data, we set the data as the value of the input, and it will instantly update where we are calling the name value:

```
<p>{{ name }}</p>
```

Go ahead and run npm run dev from the command line to compile it down:

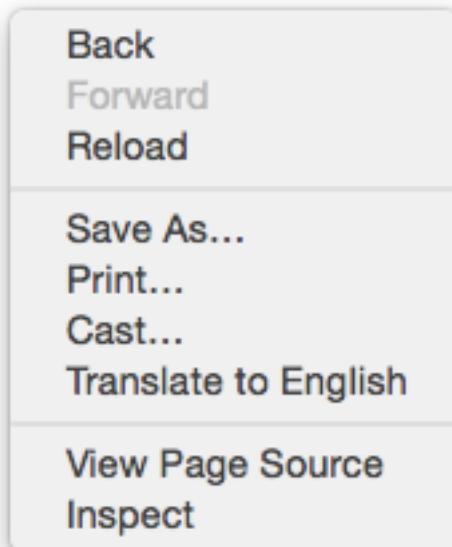
```
npm run dev
```

And now on your /pages index page, you should see the input screen. Enter in your name and watch how it is instantly displayed below it:

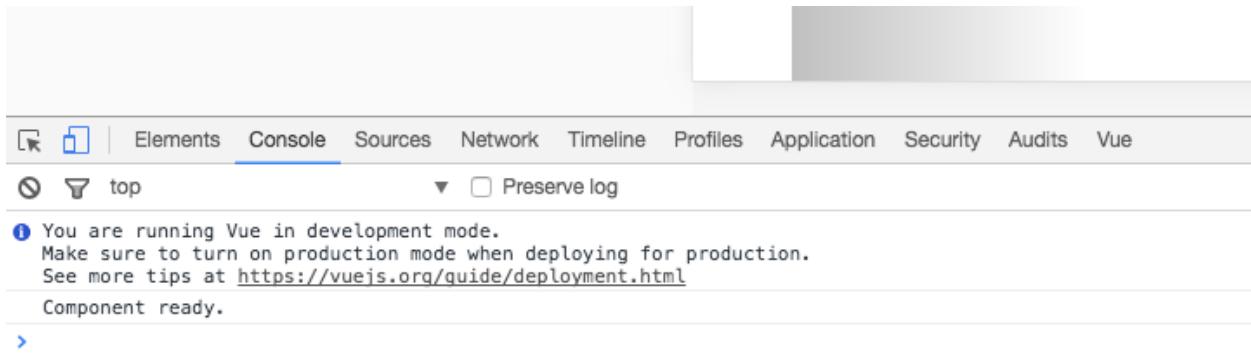


That's pretty simple stuff.

If it doesn't work, use the console feedback to trouble shoot the problem:



Select inspect, which will split the screen and default to the elements tab. Select the console tab:



Ok, back to Example.vue. We have a mounted method that currently logs to the console. Typically, we would use this to call a method that in turn would populate some initial data into our model.

Let's make the following change to Example.vue:

Gist:

[loadData method](#)

From book: _____

```
<script>

  var givenName = 'Bill';

  export default {

    data : function(){

      return {

        name: ''


      },


    },


    methods : {

      loadData : function(){

        this.name = givenName;

      }

    },


    mounted() {

      this.loadData();

      console.log('Component ready.')

    }

  }

</script>
```

So first, I've created a variable to simulate data that might come from an outside source, like an ajax call:

```
var givenName = 'Bill';
```

Next you can see that we have a new object in our component named methods, and this is where we define methods we want to use.

In this case, we named it loadData:

```
methods : {

  loadData : function(){

    this.name = givenName;

  }

},
```

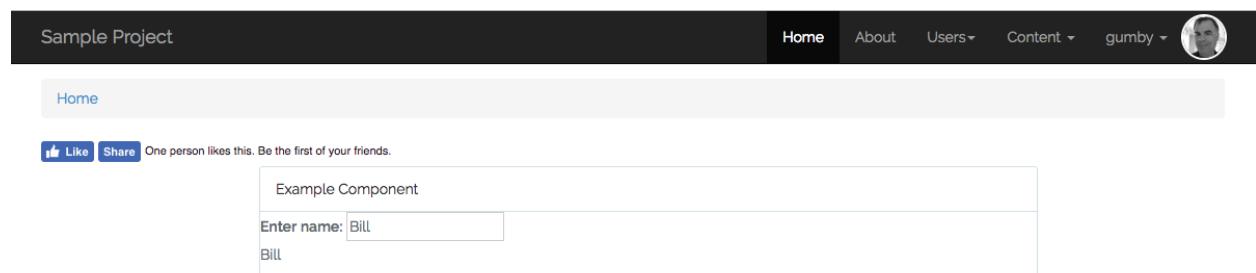
And you can see that just sets the name in our data object to the givenName value.

Finally, we call the loadData method from our mounted method:

```
mounted() {

  this.loadData();
  console.log('Component ready.')
}
```

Go ahead and run npm run dev and then go to your /pages route. Now you will see the component loads with the value we set in the data:



The form is still interactive, it will update as you overwrite the default.

I'm using super simple examples here to give you some idea of two way model binding.

Ok, so now we'll add a clearData method and button. We have to change both the template and the script, so I will give you the entire file:

Gist:

[@click and clear method](#)

From book:

```
<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 id="name-form">
            <label for="name">Enter name:</label>
            <input type="text" v-model="name" id="name" name="name" />
            <p>{{ name }}</p>
            <button @click="clearData()" class="btn btn-default">
              Clear
            </button>
          </div>
        </div>
      </div>
    </div>
  </div>
</template>

<script>
  var givenName = 'Bill';

  export default {
    data : function(){
      return {
        name: ''
      }
    }
  }
</script>
```

```
        },  
  
        methods : {  
  
            loadData : function(){  
  
                this.name = givenName;  
  
            },  
  
            clearData : function(){  
  
                this.name = '';  
            }  
  
        },  
        mounted() {  
  
            this.loadData();  
  
            console.log('Component ready.')  
  
        }  
    }  
  
</script>
```

So in the template, I added a button:

```
<button @click="clearData()" class="btn btn-default">  
    Clear  
</button>
```

You can see the @click directive and it is calling the clearData method, which we have added to our script:

```
methods : {  
  
    loadData : function(){  
  
        this.name = givenName;  
  
    },  
  
    clearData : function(){  
  
        this.name = '';  
    }  
},
```

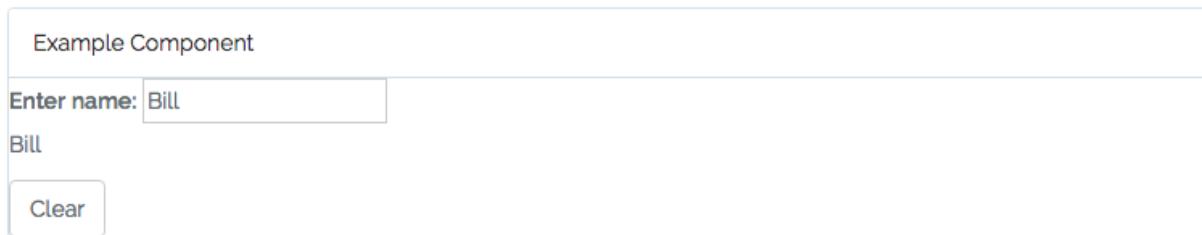
All we are doing there is setting this.name to an empty string. So go ahead and run npm run dev and you will see the following:

Example Component

Enter name:

Bill

[Clear](#)



And obviously, if you hit the clear button, it will clear the form.

The v-for directive is likely one you will use a lot, so here is another super simple example:

Gist:

[v-for](#)

From book:

```
<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">List the Beatles</div>
          <div><ul v-for="member in beatles">
            <li>{{member}}</li>
          </ul>
        </div>
      </div>
    </div>
  </div>
</template>

<script>

  var bandMembers =[ 'John' , 'Paul' , 'George' , 'Ringo' ];

  export default {

    data : function(){

      return {

        beatles: []
      }
    },
    methods : {

      loadData : function(){

        this.beatles = bandMembers;

      }
    },
    mounted() {

```

```
    this.loadData();
    console.log('Component ready.')

}

</script>
```

Ok, so on the template, we have a div:

```
<div>

<ul v-for="member in beatles">

<li>{{member}}</li>

</ul>

</div>
```

The v-for directive creates a local variable, which I've named 'member,' which holds a value from the beatles array, just like a foreach loop in PHP, but the local variable comes first, not the object.

Then we just call the variable in the list item and we get the value.

Of course we have to load data into the beatles array, and we will do that in the <script>. We start by declaring the bandMembers variable:

```
var bandMembers = ['John', 'Paul', 'George', 'Ringo'];
```

Next we create a beatles property of data:

```
data : function(){  
  
    return {  
        beatles: []  
  
    }  
}
```

And finally we add to our loadData method:

```
loadData : function(){  
  
    this.beatles = bandMembers;  
  
},
```

So you can see we are loading in the data from bandMembers and setting to this.beatles.

Go ahead and run npm run dev and you will see:



The screenshot shows a web application interface. At the top, there is a dark header bar with the text "Sample Project" on the left and navigation links "Home", "About", "Users", "Content", and "gumby" on the right. A user profile picture is also visible. Below the header, there is a light-colored main content area. In the top left of this area, there are "Like" and "Share" buttons, followed by the text "One person likes this. Be the first of your friends.". Below this, there is a section titled "List the Beatles" containing a bulleted list of names: "John", "Paul", "George", and "Ringo".

So you are getting that result from <example></example>, which is now a reusable component, anywhere in your views that you want to use it.

Obviously, this is just a baby-step example to get you up and running in Vue.js. I'm going to list the resources here again for further study:

[Vue.js](#)

[The Majesty of Vue](#)

Summary

This chapter got you up and running with Mix, which helps us integrate powerful tools into our presentation. One of those tools is Vue.js.

I would recommend doing some study outside of this book on Vue.js, since this chapter was meant to be an introduction and not a comprehensive tutorial on Vue. That subject is worthy of its own book, and I've recommended [The Majesty of Vue](#), if you want to dig in deep with Vue.

In the next chapter, it's going to get a lot more complicated, more of intermediate level, since we are going to cover datagrids, powered by ajax and Vue.

Chapter 12: Data Grids with Vue.js

So last in the chapter we got introduced to Vue.js, a powerful javascript framework that will help us build datagrids that feature searchable, sortable columns with pagination. In this chapter, we are going to build datagrids, using ajax and Vue.js.

Datagrid

While Laravel's pagination method provides us with a simple and quick way to display pages of results, we need a deeper implementation if we want things like search and column sorts. And since these features are so common as requirements, we will build up a nice implementation.

Api Route

The data for our datagrids is going to come from an ajax call, and because of that, we need to specify a route in routes/web.php:

```
// Api Routes  
  
Route::get('api/widget-data', 'ApiController@widgetData');
```

Now you may ask why are we not using the api.php file? The reason is that those routes require us to setup api authentication, which is not a topic we are covering in this book, and it's not necessary for an ajax call inside the application. It's meant more for calls from outside the app.

ApiController

Logically, the next step is to create an ApiController:

```
php artisan make:controller ApiController
```

Ok, great, now we're ready to set this up to call some data. Let's add to the use statement:

```
use App\Widget;
```

Next, let's add the following method to the ApiController:

Gist:

[widgetData method](#)

From book:

```

public function widgetData()
{
    $rows = Widget::select('id as Id',
        'name as Name',
        'created_at as Created')
        ->paginate(10);

    return response()->json($rows);
}

```

We're using the query builder select method to return results, with the format specified, for example id as Id. That will get us uppercase column names.

We are also return a response formatted in json, which is how we want to consume the data in our ajax call. To check out the results, try the following url:

`sample-project.com/api/widget-data`

I've formatted the results for readability, but you should get the same results, just not formatted with line breaks:

```

{
    "total": 31,
    "per_page": 10,
    "Current_page": 1,
    "Last_page": 4,
    "Next_page_url": "http://sample-project.com/api/widget-data?page=2",
    "Prev_page_url": null,
    "From": 1,
    "to": 10,
    "Data": [
        {"Id": 1, "Name": "sed repellendus", "Created": "2016-08-21 04:05:50"}, ,
        {"Id": 2, "Name": "incidunt recusandae", "Created": "2016-08-21 04:05:50"}, ,
        {"Id": 3, "Name": "asperiores pariatur", "Created": "2016-08-21 04:05:50"}, ,
        {"Id": 4, "Name": "ullam animi", "Created": "2016-08-21 04:05:50"}, ,
        {"Id": 5, "Name": "illum similique", "Created": "2016-08-21 04:05:50"}, ,
        {"Id": 6, "Name": "omnis qui", "Created": "2016-08-21 04:05:50"}, ,
        {"Id": 7, "Name": "tempore et", "Created": "2016-08-21 04:05:50"}, ,
        {"Id": 8, "Name": "qui ut", "Created": "2016-08-21 04:05:50"}, ,
        {"Id": 9, "Name": "et est", "Created": "2016-08-21 04:05:50"}, ,
        {"Id": 10, "Name": "autem et", "Created": "2016-08-21 04:05:50"}, ,
        {"Id": 11, "Name": "et et", "Created": "2016-08-21 04:05:50"}, ,
        {"Id": 12, "Name": "et et", "Created": "2016-08-21 04:05:50"}, ,
        {"Id": 13, "Name": "et et", "Created": "2016-08-21 04:05:50"}, ,
        {"Id": 14, "Name": "et et", "Created": "2016-08-21 04:05:50"}, ,
        {"Id": 15, "Name": "et et", "Created": "2016-08-21 04:05:50"}, ,
        {"Id": 16, "Name": "et et", "Created": "2016-08-21 04:05:50"}, ,
        {"Id": 17, "Name": "et et", "Created": "2016-08-21 04:05:50"}, ,
        {"Id": 18, "Name": "et et", "Created": "2016-08-21 04:05:50"}, ,
        {"Id": 19, "Name": "et et", "Created": "2016-08-21 04:05:50"}, ,
        {"Id": 20, "Name": "et et", "Created": "2016-08-21 04:05:50"}, ,
        {"Id": 21, "Name": "et et", "Created": "2016-08-21 04:05:50"}, ,
        {"Id": 22, "Name": "et et", "Created": "2016-08-21 04:05:50"}, ,
        {"Id": 23, "Name": "et et", "Created": "2016-08-21 04:05:50"}, ,
        {"Id": 24, "Name": "et et", "Created": "2016-08-21 04:05:50"}, ,
        {"Id": 25, "Name": "et et", "Created": "2016-08-21 04:05:50"}, ,
        {"Id": 26, "Name": "et et", "Created": "2016-08-21 04:05:50"}, ,
        {"Id": 27, "Name": "et et", "Created": "2016-08-21 04:05:50"}, ,
        {"Id": 28, "Name": "et et", "Created": "2016-08-21 04:05:50"}, ,
        {"Id": 29, "Name": "et et", "Created": "2016-08-21 04:05:50"}, ,
        {"Id": 30, "Name": "et et", "Created": "2016-08-21 04:05:50"}, ,
        {"Id": 31, "Name": "et et", "Created": "2016-08-21 04:05:50"}]
}

```

```
{"Id":7,"Name":"vel amor","Created":"2016-08-21 04:05:50"},  
 {"Id":8,"Name":"in numquam","Created":"2016-08-21 04:05:50"},  
 {"Id":9,"Name":"voluptate soluta","Created":"2016-08-21 04:05:50"},  
 {"Id":10,"Name":"quia non","Created":"2016-08-21 04:05:50"}]
```

You can see that not only did we get the rows we wanted, but we also got the pagination meta data, which we will use to construct our grid navigation.

Note: There's a free Chrome plugin from callumlocke.co.uk that you can get from the chrome store that will format your raw json results to make them more readable. Just google:

```
chrome store json formatter callumlocke.co.uk
```

Looking at our json results, we notice we have a problem. We are not getting the benefit of the accessor we created in SuperModel, which we extended to Widget. The hours, minutes, and seconds are appearing in our Created value and we don't want that.

So instead of using an Eloquent model call, we are going to use a raw DB query, which will be easier to manipulate. And we will also be able to continue to use many of the fluent query builder methods. Laravel is just awesome for this.

So let's remove AppWidget from the use statement on the ApiController and replace it with:

```
Use DB;
```

Then let's change the widgetData method to the following:

Gist:

[WidgetData Method](#)

From book:

```

public function widgetData()
{
    $rows = DB::table('widgets')
        ->select('id as Id',
                  'name as Name',
                  DB::raw('DATE_FORMAT(created_at, "%m-%d-%Y") as Created'))
        ->paginate(10);

    return response()->json($rows);
}

```

So this time we are using Laravel's fluent query builder, which has methods in common with eloquent like the pagination method.

Note that in the DB:: syntax, we use the name of the table, which is 'widgets' in this case. Within the select statement, we have nested a DB::raw expression:

```
DB::raw('DATE_FORMAT(created_at, "%m-%d-%Y") as Created'))
```

And this is telling us we want the DATE_FORMAT value of the result, so now our json output produces:

```
{"total":31,
"Per_page":10,
"Current_page":1,
"Last_page":4,
"Next_page_url":
"http://sample-project.com/api/widget-data?page=2",
"Prev_page_url":null,
"From":1,"to":10,
>Data":
[{"Id":1,"Name":"sed repellendus","Created":"08-21-2016"}, {"Id":2,"Name":"incidunt recusandae","Created":"08-21-2016"}, {"Id":3,"Name":"asperiores pariatur","Created":"08-21-2016"}, {"Id":4,"Name":"ullam animi","Created":"08-21-2016"}, {"Id":5,"Name":"illum similique","Created":"08-21-2016"}, {"Id":6,"Name":"omnis qui","Created":"08-21-2016"}, {"Id":7,"Name":"vel amor","Created":"08-21-2016"}, {"Id":8,"Name":"in numquam","Created":"08-21-2016"}, {"Id":9,"Name":"voluptate soluta","Created":"08-21-2016"}, {"Id":10,"Name":"quia non","Created":"08-21-2016"}]}
```

So far, we haven't done anything that would help us sort columns or search by keyword, and yet our method is taking up 7 lines of code, depending on whether or not you count the blank lines left in for readability.

There's nothing necessarily wrong with this, but this method is going to grow, since we have to add conditions to the query. And if you imagine that you might have numerous methods in your ApiController, it can get unwieldy very quickly. We don't want that.

It would be better if we can reduce the controller method down to something like:

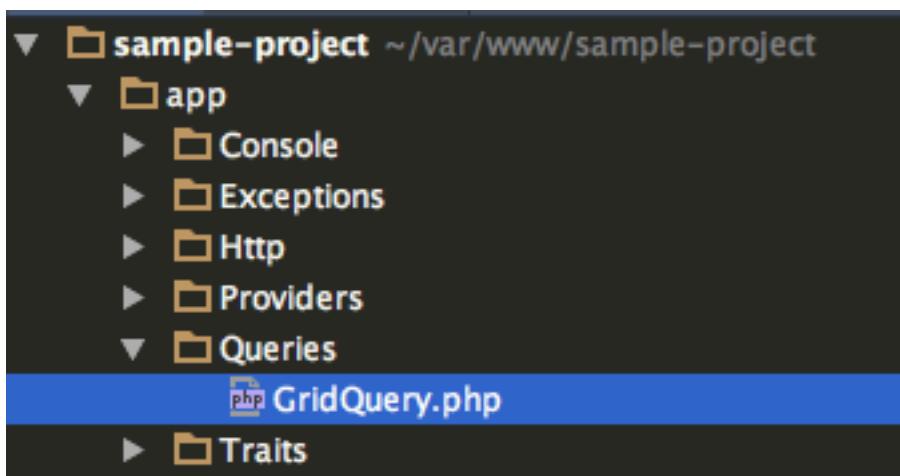
```
public function widgetData(Request $request)
{
    return GridQuery::sendData($request, 'widgets');
}
```

I'm assuming in this case that I'm probably going to build an application with more than one datagrid. So it really helps us avoid code duplication if we can do this in an extensible way.

You could go the repository route and create a specific repository for each datagrid you want to build. That would work, but you might end up repeating code in each repository for things like the column sorts and keyword filtering.

There are no hard rules on how to do this, so you have a lot of flexibility. On a recent episode of Laracasts, Jeffrey Way talked about Query objects, and that approach resonates with me.

So let's see if we can build this up. Let's create a folder in your app directory named Queries, and within that create a php file named GridQuery:



Let's add the following to the GridQuery file:

Gist:

[GridQuery.php](#)

From book:

```
namespace App\Queries;

use Illuminate\Http\Request;
use DB;

class GridQuery
{

    public static function sendData(Request $request, $table)
    {

        $rows = DB::table($table)
            ->select('id as Id',
                      'name as Name',
                      DB::raw('DATE_FORMAT(created_at, "%m-%d-%Y") as Created'))
            ->paginate(10);

        return response()->json($rows);

    }

}
```

Ok, so our sendData method is just the same query that we had before. This is not a permanent solution, we'll just use this as a test to make sure we can still pull the data.

So in your ApiController, change the widgetData method to the following:

```
public function widgetData(Request $request)
{
    return GridQuery::sendData($request, 'widgets');
}
```

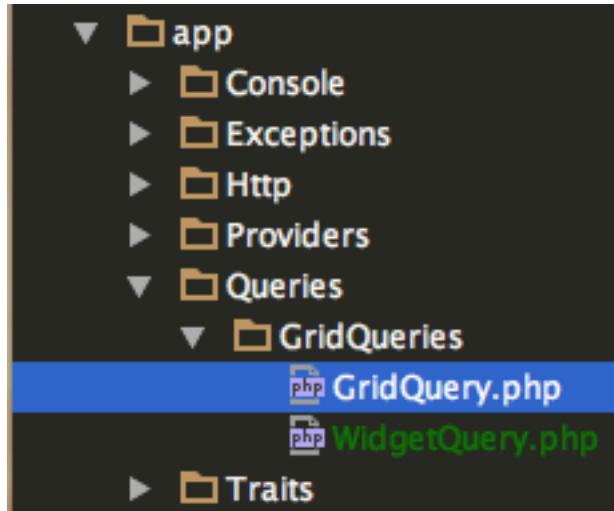
Before we forget, let's chop the use DB statement and add the use statement for GridQuery at the top of the file:

```
use App\Queries\GridQuery;
```

Ok, so if you try that now, you should get the results that we had before.

If you are wondering why we are pulling in an instance of Request, it's because we are going to use it later when we hand in the column sort and keyword search requests. But we're not ready to work on that just yet.

To make this a little more flexible, we need to modify the folder structure a little. Let's create a GridQueries folder inside of the queries folder:



We are also going to create a WidgetQuery.php file in the same folder, so let's go ahead and do that now.

Before we go on, I'm going to give you the three files we are working with, just to avoid confusion, since we have changed things. Let's start with GridQuery.php, which is now in the GridQueries folder:

Gist:

[GridQuery](#)

From book:

```
<?php

namespace App\Queries\GridQueries;

use Illuminate\Http\Request;
use DB;

class GridQuery
{

    public static function sendData(Request $request, $query)
    {

        return response()->json($query->data($request));
    }

}
```

So the thing to keep in mind here is that for the \$query variable, we are going to hand in an instance of WidgetQuery, which we will call in the signature of sendData method when we use it in the ApiController class.

Let's take a look at that next:

Gist:

[ApiController](#)

From book:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Queries\GridQueries\GridQuery;
use App\Queries\GridQueries\WidgetQuery;
use App\Http\Requests;
```

```
class ApiController extends Controller
{
    public function widgetData(Request $request)
    {
        return GridQuery::sendData($request, new WidgetQuery);
    }
}
```

You can see new WidgetQuery is being called, so let's create that in WidgetQuery.php with the following:

Gist:

[WidgetQuery.php](#)

From book:

```
<?php

namespace App\Queries\GridQueries;
use DB;
use Illuminate\Http\Request;

class WidgetQuery
{
    public function data(Request $request)
    {
        $rows = DB::table('widgets')
            ->select('id as Id',
                      'name as Name',
                      DB::raw('DATE_FORMAT(created_at, "%m-%d-%Y") as Created'))
            ->paginate(10);

        return $rows;
    }
}
```

```
 }  
}  


---


```

If you were wondering where the query was, there it is.

Ok, this seems like a lot of work to replace a small amount of query code, but like I said, the old approach did not scale well. If you can imagine a different query class being used, new MarketingImageQuery, for example, everything will be exactly the same, except for the calling the class like so:

```
return GridQuery::sendData($request, new MarketingImageQuery);
```

This should give you a sense of how we intend to use this. Now we are not going to make that query class just yet, but we will in the future.

The main point to get from this is that we have reduced the redundant code fairly effectively. Also, the syntax is helpful, we just check to see what GridQuery::sendData does:

```
public static function sendData(Request $request, $query)  
{  
    return response()->json($query->data($request));  
}
```

You see it takes in the request, which we have not used yet, and the query instance, which could be WidgetQuery for example. We return the response in json, feeding in the instance of the query class's data method, passing along the \$request.

Each query class that we build to operate in our GridQuery::sendData method will have a data method, and that is what will return the actual data that we need. Hopefully all that is clear.

You can go to the following url and see it all works as expected:

sample-project.com/api/widget-data

Now one of the big reasons we did all this is that our filtering and sorting of the data is going to make the files bigger. But the ApiController, will still only have a one line method to return the data, which is what we wanted.

Now we need to make some big changes to our two files. Let's start with GridQuery.php:

Gist:

[GridQuery](#)

From book:

```
<?php

namespace App\Queries\GridQueries;

use Illuminate\Http\Request;
use DB;

class GridQuery
{

    public static function sendData(Request $request, $query)
    {

        // set sort by column and direction

        list($column, $direction) = static::setSort($request);

        // search by keyword with column sort

        if ($request->has('keyword')) {

            return static::keywordFilter($request, $query, $column, $direction);

        }

        // return data
```

```
    return static::getData($query, $column, $direction);

}

public static function setSort(Request $request)
{
    // set sort by column with default

    $column = 'id';
    $direction = 'asc';

    if ($request->has('column')) {

        $column = $request->get('column');

        if ($column == 'Id') {

            $direction = $request->get('direction') == 1 ? 'asc' : 'desc';

            return [$column, $direction];
        } else {

            $direction = $request->get('direction') == 1 ? 'desc' : 'asc';

            return [$column, $direction];
        }
    }

    return [$column, $direction];
}

public static function keywordFilter(Request $request,
                                    $query, $column,
                                    $direction)
{
    $keyword = $request->get('keyword');

    return response()->json($query->filteredData($column,
```

We will step through that, but before we do, let's modify `WidgetQuery.php` to the following:

Gist:

WidgetQuery.php Modified

<?php

```
namespace App\Queries\GridQueries;
use DB;
use Illuminate\Http\Request;

class WidgetQuery
{
    public function data($column, $direction)
    {
        $rows = DB::table('widgets')
            ->select('id as Id',
                      'name as Name',
                      'slug as Slug',
                      DB::raw('DATE_FORMAT(created_at,
                                         "%m-%d-%Y") as Created'))
            ->orderBy($column, $direction);
        return $rows;
    }
}
```

```
        ->orderBy($column, $direction)
        ->paginate(10);

    return $rows;

}

public function filteredData($column, $direction, $keyword)
{
    $rows = DB::table('widgets')
        ->select('id as Id',
                  'name as Name',
                  'slug as Slug',
                  DB::raw('DATE_FORMAT(created_at,
                      "%m-%d-%Y") as Created'))
        ->where('name', 'like', '%' . $keyword . '%')
        ->orderBy($column, $direction)
        ->paginate(10);

    return $rows;
}

}
```

Ok, you can see we modified the data method to take in a column and direction. This allows us to add the fluent method of orderBy, where we give it the column and direction. If those were not handed in via request, they will default to id and asc.

We added an entirely new method, filteredData:

```

public function filteredData($column, $direction, $keyword)
{
    $rows = DB::table('widgets')
        ->select('id as Id',
                  'name as Name',
                  'slug as Slug',
                  DB::raw('DATE_FORMAT(created_at,
                      "%m-%d-%Y") as Created'))
        ->where('name', 'like', '%' . $keyword . '%')
        ->orderBy($column, $direction)
        ->paginate(10);

    return $rows;
}

```

In this method, we get the `->where()` method where name is like the keyword.

Now when you build a query class for different queries, they may be more complex, you might have relationships that are eager loaded, etc. So you can see how effective this structure will be for you. You will only need to write the custom queries, the `GridQuery` class will not need to change. And of course this makes it relatively easy to standardize our Vue component, which is going to send the ajax request that will call the `ApiController`'s `widgetData` method.

Before we move on to that part, let's look at `GridQuery.php`, since it's doing so much more now than it was before. Let's look at the `sendData` method:

```

public static function sendData(Request $request, $query)
{
    // set sort by column and direction

    list($column, $direction) = static::setSort($request);

    // search by keyword with column sort

    if ($request->has('keyword')) {

```

```
    return static::keywordFilter($request,
                                 $query,
                                 $column,
                                 $direction

    );

}

// return data

return static::getData($query, $column, $direction);

}
```

We start by listing the results of the setSort method:

```
public static function setSort(Request $request)
{
    // set sort by column with default

    $column = 'id';
    $direction = 'asc';

    if ($request->has('column')) {

        $column = $request->get('column');

        if ($column == 'Id') {

            $direction = $request->get('direction') == 1 ? 'asc' : 'desc';

            return [$column, $direction];
        } else {
    
```

```

    $direction = $request->get('direction') == 1 ? 'desc' : 'asc';

    return [$column, $direction];
}

}

return [$column, $direction];
}

```

We set a default, so if the \$request has no column or direction to hand in, we still have a default sort order. Then we have an if statement to check if the column we are to sort on is ID, and if so, we set it to the opposite direction. Since it defaults to ascending, it makes sense to have it be descending when the column is clicked. Then we return the two values, which are then listed in the sendData method:

```
list($column, $direction) = static::setSort($request);
```

Next we check to see if we have a keyword in the request:

```

if ($request->has('keyword')) {

return static::keywordFilter($request, $query, $column, $direction);

}

```

If it has a keyword, we get the keywordFilter method, which hands the parameters we need to:

```

return response()->json($query->filteredData($column,
                                                 $direction,
                                                 $keyword

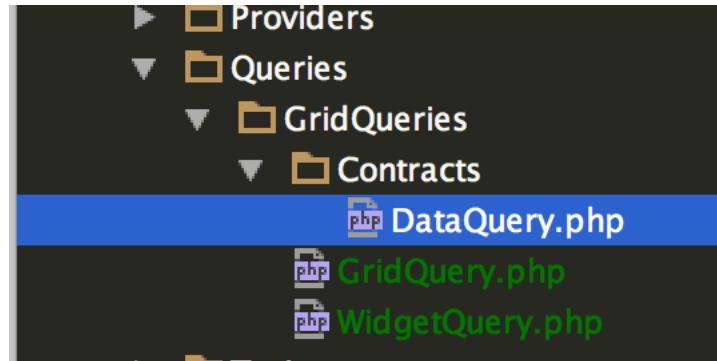
));

```

And again, the GridQuery class will assume the query class has a filteredData method. In fact, why don't we implement a contract to enforce this?

Implementing A Contract

Let's create a Contracts folder in your GridQueries folder and a DataQuery.php file in that folder:



Let's place the following in DataQuery.php:

Gist:

[DataQuery.php](#)

From book:

```
<?php
```

```
namespace App\Queries\GridQueries\Contracts;

interface DataQuery
{
    public function data($column, $direction);
    public function filteredData($column, $direction, $keyword);
}
```

So all we are doing here is forcing any class that implements this contract to have a data method and filteredData method as shown above.

Now on GridQuery.php, we have to modify the sendData method:

```
public static function sendData(Request $request, DataQuery $query)  
{
```

Now \$query is an instance of DataQuery. Don't forget to add the use statement for DataQuery to GridQuery.php:

```
use App\Queries\GridQueries\Contracts\DataQuery;
```

Next we need to change WidgetQuery to implement DataQuery:

```
class WidgetQuery implements DataQuery  
{
```

This is going to force WidgetData to have the methods listed on the DataQuery interface.

You also need the use statement here as well:

```
use App\Queries\GridQueries\Contracts\DataQuery;
```

So now, in the future, if you forget to implement DataQuery on the classes you want to hand into GridQuery::sendData(), you will get an error message.

You don't have to do this, but it is nice documentation for what your query object should have on it and will help you in debugging.

If you want to play around with the queries before we move on to the Vue implementation, you can try some variations in the browser like so:

```
http://sample-project.com/api/widget-data?column=name&direction=0&keyword=a
```

And with that, we are ready to move on to our Vue.js implementation of our widget data grid. Let's start with a couple of clean up tasks.

In your WidgetController, change the index method to the following:

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

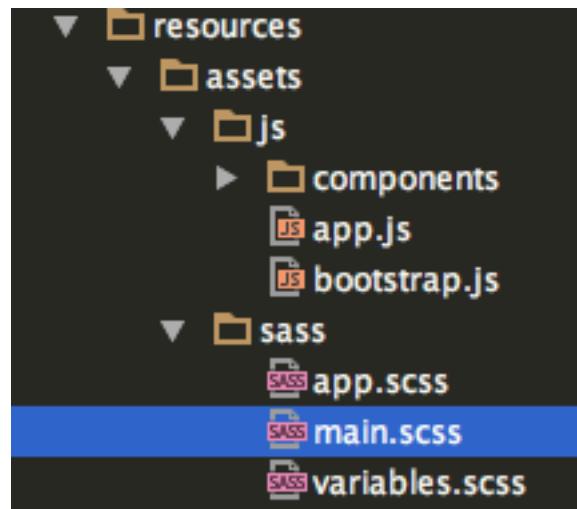
}
```

At this point in the book, there should be no gist needed for that. So obviously we are not querying for records in the index method, since we will do it all through ajax.

main.scss

Right now we have added some of our own css to app.scss in resources/assets/sass, but since we are going to be adding to that, it makes more sense to create another file to keep things more organized and separated.

Create the new file in your sass folder like so:



In your main.scss file, place the following:

Gist:

[main.scss](#)

From book:

```
body{  
  
padding-top: 65px;  
background-color: white;  
  
}  
  
.circ{  
  
width : 40px;  
vertical-align: middle;  
border-radius: 50%;  
box-shadow: 0 0 0 2px #fff;  
margin-top: 5px;  
  
}  
  
.circ:hover{  
  
box-shadow: 0 0 0 3px #f00;  
  
}  
  
.carousel-inner img {  
  
margin: auto;  
  
}  
  
th.active .arrow {  
  
opacity: 1;  
  
}  
  
.arrow {  
  
display: inline-block;  
vertical-align: middle;  
width: 0;  
height: 0;  
margin-left: 5px;  
opacity: 0.66;
```

```
}

.arrow.asc {
    border-left: 4px solid transparent;
    border-right: 4px solid transparent;
    border-bottom: 4px solid black;
}

.arrow.dsc {
    border-left: 4px solid transparent;
    border-right: 4px solid transparent;
    border-top: 4px solid black;
}

table thead tr th {
    cursor: pointer;
}

.for-page-button {
    margin-top: 20px;
    margin-left: 20px;
}

.number-input {
    width: 60px;
}
```

Now let's chop out the styles from app.scss and add a call to main.scss, so we are just left with:

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

// Variables
@import "variables";

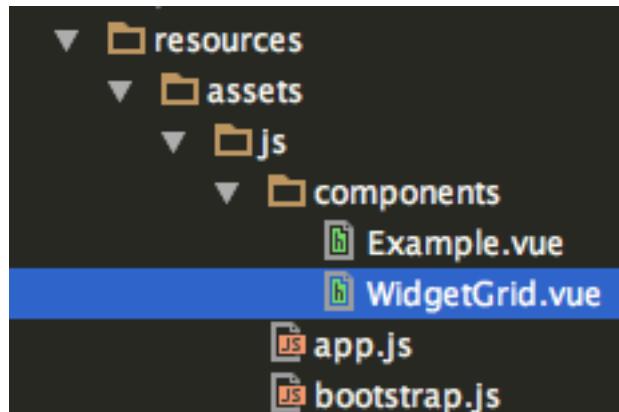
// Bootstrap
@import "node_modules/bootstrap-sass/assets/stylesheets/bootstrap";

// bring in main
@import "main";
```

Ok, let's move onto app.js in resources/assets/js and add the following line below the call to the example component:

```
Vue.component('example', require('./components/Example.vue'));
Vue.component('widget-grid', require('./components/WidgetGrid.vue'));
```

We don't have WidgetGrid.vue yet, so let's make that now:



In that file, place the following:

Gist:

[WidgetGrid.vue](#)

From book:

```
<template>

<div class="row">

<div class="col-lg-12">

<form id="search">

    Search

    <input name="query"
           v-model="query"
           @keyup="search(query)">

</form>

<div class="pull-right">

    {{ total }} Total Results

</div>

<section class="panel">
<div class="panel-body">

<table class="table table-bordered table-striped">

    <thead>

        <tr>

            <th v-for="key in gridColumnns"
                @click="sortBy(key)"
                v-bind:class="{ active: sortKey == key }">

                {{ key }}

                <span class="arrow"
                      v-bind:class="sortOrder > 0 ? 'asc' : 'dsc'">
                </span>

            </th>

        <th>
```

```
Actions

</th>

</tr>

</thead>

<tbody>

<tr v-for="row in gridData">

  <td>

    {{ row.Id }}

  </td>

  <td>

    <a v-bind:href="/widget/' + row.Id + '-' + row.Slug">

      {{ row.Name }}

    </a>

  </td>

  <td>

    {{ row.Created }}

  </td>

  <td><a v-bind:href="/widget/' + row.Id + '/edit'">

    <button type="button"
            class="btn btn-default">

      Edit

    </button>

  </td>
```

```
</a>

</td>

</tr>

</tbody>

</table>

</div>

<div class="pull-right">

    page {{ current_page }} of {{ last_page }} pages

</div>

</section>

<div class="row">

    <div class="pull-right for-page-button">

        <button @click="getData(go_to_page)"
                class="btn btn-default">

            Go To Page:

        </button>

        <input v-model="go_to_page" class="number-input"></input>

    </div>

    <ul class="pagination pull-right">

        <li><a @click.prevent="getData(first_page_url)">

            first

        </a>

    </li>
```

```
<li v-if="checkUrlNotNull(prev_page_url)">

  <a @click.prevent="getData(prev_page_url)">

    prev

  </a>

</li>

<li v-for="page in pages"
  v-if="page >
  current_page - 2 && page < current_page + 2"
  v-bind:class="{ 'active': checkPage(page) }">

  <a @click.prevent="getData(page)">

    {{ page }}

  </a>

</li>

<li v-if="checkUrlNotNull(next_page_url)">

  <a @click.prevent="getData(next_page_url)">

    next

  </a>

</li>

<li>

  <a @click.prevent="getData(last_page_url)">

    last

  </a>

</li>
```

```
</ul>

</div>

</div>

</div>

</template>

<script>

export default {

  mounted: function () {

    this.loadData();
  },

  data: function () {

    return {

      query: '',
      gridColumnns: ['Id', 'Name', 'Created'],
      gridData: [],
      total: null,
      next_page_url: null,
      prev_page_url: null,
      last_page: null,
      current_page: null,
      pages: [],
      first_page_url: null,
      last_page_url: null,
      go_to_page: null,
      sortOrder: 1,
      sortKey: ''
    }
  },
}
```

```
methods: {

    sortBy: function (key){
        this.sortKey = key;
        this.sortOrder = (this.sortOrder == 1) ? -1 : 1;
        this.getData(1);

    },

    search: function(query){

        this.getData(query);

    },

    loadData: function (){

        $.getJSON('api/widget-data', function (data) {

            this.gridData = data.data;
            this.total = data.total;
            this.last_page = data.last_page;
            this.next_page_url = data.next_page_url;
            this.prev_page_url = data.prev_page_url;
            this.current_page = data.current_page;
            this.first_page_url = 'api/widget-data?page=1';
            this.last_page_url = 'api/widget-data?page='
                + this.last_page;
            this.setPageNumbers();

        }.bind(this));
    },

    setPageNumbers: function(){

        for (var i = 1; i <= this.last_page; i++) {

            this.pages.push(i);

        }

    },
}
```

```
getData: function (request){  
  
    let getPage;  
  
    switch (request){  
  
        case this.prev_page_url :  
  
            getPage = this.prev_page_url +  
                '&column=' + this.sortKey +  
                '&direction=' + this.sortOrder;  
  
            break;  
  
        case this.next_page_url :  
  
            getPage = this.next_page_url +  
                '&column=' + this.sortKey +  
                '&direction=' + this.sortOrder;  
  
            break;  
  
        case this.first_page_url :  
  
            getPage = this.first_page_url +  
                '&column=' + this.sortKey +  
                '&direction=' + this.sortOrder;  
  
            break;  
  
        case this.last_page_url :  
  
            getPage = this.last_page_url +  
                '&column=' + this.sortKey +  
                '&direction=' + this.sortOrder;  
  
            break;  
  
        case this.query :  
  
            getPage = 'api/widget-data?' +  
                'keyword=' + this.query +
```

```
'&column=' + this.sortKey +
'&direction=' + this.sortOrder;

break;

case this.go_to_page :

if( this.go_to_page != '' && this.pageInRange()){

getPage = 'api/widget-data?' +
'page=' + this.go_to_page +
'&column=' + this.sortKey +
'&direction=' + this.sortOrder +
'&keyword=' + this.query;

this.clearPageNumberInputBox();

} else {

alert('Please enter a valid page number');

}

break;

default :

getPage = 'api/widget-data?' +
'page=' + request +
'&column=' + this.sortKey +
'&direction=' + this.sortOrder +
'&keyword=' + this.query;

break;

}

if (this.query == '' && getPage != null){

$.getJSON(getPage, function (data) {

this.gridData = data.data;
this.total = data.total;
this.last_page = data.last_page;
```

```
        this.next_page_url = data.next_page_url;
        this.prev_page_url = data.prev_page_url;
        this.current_page = data.current_page;

    }.bind(this));

} else {

    if (getPage != null){

        $.getJSON(getPage, function (data) {

            this.gridData = data.data;
            this.total = data.total;
            this.last_page = data.last_page;
            this.next_page_url = (data.next_page_url == null) ?
                null : data.next_page_url + '&keyword=' +this.query;
            this.prev_page_url = (data.prev_page_url == null) ?
                null : data.prev_page_url + '&keyword=' +this.query;
            this.first_page_url =
                'api/widget-data?page=1&keyword=' + this.query;
            this.last_page_url = 'api/widget-data?page='
                + this.last_page + '&keyword=' +this.query;
            this.current_page = data.current_page;
            this.resetPageNumbers();

        }.bind(this));

    }

},
checkPage: function(page){

    return page == this.current_page;

},
resetPageNumbers: function(){

    this.pages = [];

}
```

```
        for (var i = 1; i <= this.last_page; i++) {
            this.pages.push(i);
        }

    },
checkUrlNotNull: function(url){
    return url != null;
},
clearPageNumberInputBox: function(){
    return this.go_to_page = '';
},
pageInRange: function(){
    return this.go_to_page <= parseInt(this.last_page);
}
}
}

</script>
```

Definitely use the Gist for code formatting on this one.

From the command line, run:

```
npm run dev
```

Everything should compile fine, if not check for typos.

Ok, let's go to resources/views/widget/index.blade.php and change it to the following:

Gist:

[index.blade.php](#)

From book:

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

@section('title')
    <title>Widgets</title>

@endsection

@section('content')
    <ol class='breadcrumb'>
        <li><a href='/'>Home</a></li>
        <li class='active'>Widgets</li>
    </ol>

    <h2>Widgets</h2>

    <hr/>

    <widget-grid></widget-grid>

    <div> <a href="/widget/create">
        <button type="button"
            class="btn btn-lg btn-primary">
            Create New
        </button>
    </a>

    </div>

@endsection
```

Now assuming everything is correct, and you have records in your DB, you will see something like the following:

The screenshot shows a web application interface for a "Sample Project". At the top, there's a dark header with the project name and navigation links for Home, About, Users, Content, and a user profile for "gumby". Below the header, a breadcrumb trail shows "Home / Widgets". The main content area is titled "Widgets" and contains a search bar. A data grid table is displayed, showing 10 rows of data with columns for "Id", "Name", "Created", and "Actions". The "Actions" column contains "Edit" buttons. A message at the top right indicates "28 Total Results". Below the grid, a pagination bar shows "page 1 of 3 pages" with buttons for "first", "1", "2", "next", and "last", along with a "Go To Page:" input field. At the bottom left, a "Create New" button is visible, and a footer notes "© 2015 - 2017 Sample Project All rights Reserved."

| Id | Name | Created | Actions |
|----|--------------------|------------|---------|
| 14 | voluptatibus omnis | 01-26-2017 | Edit |
| 25 | voluptates nihil | 01-26-2017 | Edit |
| 21 | velit molestiae | 01-26-2017 | Edit |
| 10 | ut nulla | 01-26-2017 | Edit |
| 27 | sint expedita | 01-26-2017 | Edit |
| 8 | similique sed | 01-26-2017 | Edit |
| 3 | quisquam cumque | 01-26-2017 | Edit |
| 2 | numquam etan | 01-26-2017 | Edit |
| 9 | non sunt | 01-26-2017 | Edit |
| 15 | neque voluptate | 01-26-2017 | Edit |

Ok, so play around with the grid, it's searchable, sortable, and fully paginated.

WidgetGrid.vue

Ok, we're going to break this down and learn a little more about Vue.js than we did in our baby-steps lesson. We can see we are getting a robust result from our WidgetGrid component and we need to take a closer look to see how we are accomplishing this.

As you probably know, vue component files contain the template, the script, and css for the component. In our case, since we will reuse the grid css, we have put that into main.scss instead.

I think the best way to approach this is to dive right into the <template>. Let's take a look at one of the first interesting parts, the search:

```
<form id="search">

    Search

    <input name="query"
           v-model="query"
           @keyup="search(query)">

</form>

<div class="pull-right">
    {{ total }} Total Results
</div>
```

On the search input box, we are binding the value with v-model to a variable named query. Then we use the value of query in the search method.

This is the search method in our script:

```
search: function(query){

    this.getData(query);

},
```

And you can see that passes the query into our getData method, which has a lot of parts to it. We're not ready to dig into that just yet.

Back on the input in the template, we see the Vue.js @keyup event, which is what calls the search method every time something changes in the search input.

Below that, you can see we have div that prints out a count of all results by using the 'total' property, which is part of our Vue data model:

```
<div class="pull-right">  
    {{ total }} Total Results  
</div>
```

The next interesting part is the table head:

```
<table class="table table-bordered table-striped">  
  
    <thead>  
  
        <tr>  
  
            <th v-for="key in gridColumnns"  
                @click="sortBy(key)"  
                v-bind:class="{active: sortKey == key}">  
                {{ key }}  
  
                <span class="arrow"  
                    v-bind:class="sortOrder > 0 ? 'asc' : 'dsc'">  
                </span>  
            </th>  
  
            <th>  
                Actions  
            </th>  
  
        </tr>  
  
    </thead>
```

Note that we're using a v-for directive to get the column names for the head, and we also have a <th>Actions</th> since that one is not built dynamically.

We've created a local variable named 'key' for the column headings and you can see that is what we feed into our sortBy method. The sortBy method is activated by an @click event.

Then you see something new:

```
v-bind:class="{active: sortKey == key}">
```

That binds the active css class to the <th> when it's clicked and sets the sortKey property to the selected key.

Then we have a span for the to set the arrow class, and this time we are using a ternary to determine if we want asc or dsc.

```
<span class="arrow">  
  v-bind:class="sortOrder > 0 ? 'asc' : 'dsc'">  
</span>
```

If you look in main.scss, you will find the corresponding css class.

So you can see so far, we are getting a lot of interactivity, without a lot of markup, which is why Vue is so easy to work with.

Ok, let's move on to the body of the table that holds our data:

```
<tbody>  
  <tr v-for="row in gridData">  
    <td>  
      {{ row.Id }}  
    </td>  
    <td>
```

```

<a v-bind:href="/widget/' + row.Id + '-' + row.Slug">

{{ row.Name }}

</a>

</td>

<td>

{{ row.Created }}

</td>

<td><a v-bind:href="/widget/' + row.Id + '/edit'">

<button type="button"
        class="btn btn-default">

Edit

</button>

</a>

</td>

</tr>

</tbody>

```

Again we are using the v-for directive to create a local variable, in this case it's named 'row'.

So this is how we populated the data in the table. The gridData object will retrieve the data from an ajax call on page load, we'll check that out in a minute. So all we have to do is iterate through using v-for and we can get what we want.

Here are a couple of things to keep in mind when working with Vue. You can see how we bind the href:

```
<a v-bind:href="/widget/' + row.Id + '-' + row.Slug">
```

Because it's bound, we treat it like a javascript string instead of using the double curly braces.

One more tip I probably should mention is that templates should have a div that wraps the entire component or you will get a warning message from Vue.

So our grid is a super simple example, since we only have 3 columns of data, plus one action button. Since we're coding in javascript we use row.Id to get the row data, instead of \$row->Id, which is how we would do it in PHP. Hopefully all that is clear.

The next interesting thing to look at is the page count and current page:

```
<div class="pull-right">  
    page {{ current_page }} of {{ last_page }} pages  
</div>
```

Those values are saved in the data object, so we have access to them.

Then we get our pagination:

```
<ul class="pagination pull-right">  
    <li><a @click.prevent="getData(first_page_url)">  
        first  
</a>  
</li>  
    <li v-if="checkUrlNotNull(prev_page_url)">  
        <a @click.prevent="getData(prev_page_url)">  
            prev  
</a>
```

```
</li>

<li v-for="page in pages"
    v-if="page >
        current_page - 2 && page < current_page + 2"
    v-bind:class="{ 'active': checkPage(page)}">

    <a @click.prevent="getData(page)">

        {{ page }}

    </a>

</li>

<li v-if="checkUrlNotNull(next_page_url)">

    <a @click.prevent="getData(next_page_url)">

        next

    </a>

</li>

<li>

    <a @click.prevent="getData(last_page_url)">

        last

    </a>

</li>

</ul>
```

I spread it out a little bit to make it easier to read. So we start with:

```
<li><a @click.prevent="getData(first_page_url)"> first </a></li>
```

The @click event calls the getData method with the first_page_url as the value. We add .prevent to prevent default action. You can see we are using ‘first’ for the link.

Next we have:

```
<li v-if="checkUrlNotNull(prev_page_url)">
  <a @click.prevent="getData(prev_page_url)">prev</a>
</li>
```

We use the v-if conditional to call checkUrlNotNull and feed in the prev_page_url. If it is not null, we have a previous page, and we can show the link.

This next one is a little more complicated:

```
<li v-for="page in pages"
    v-if="page > current_page - 2 && page < current_page + 2"
    v-bind:class="{ 'active': checkPage(page) }">
  <a @click.prevent="getData(page)">{{ page }}</a>
</li>
```

We only want to show page number within a range of 2, either way from the current page. You can change that by the way, if you want to display more page numbers.

To calculate this, we use v-for to set a local variable page, which we can then check the current_page property against. If it is in the range limit, it will be shown.

Since we want the active class on the current page, we use v-bind, and as a condition use the checkPage method in our script:

```
checkPage: function(page){  
  
  return page == this.current_page;  
  
},
```

We also want the page numbers to be links to that page's data:

```
<a @click.prevent="getData(page)">{{ page }}</a>
```

So that makes a call to getData using the page as the value.

The list items for next_page_url and last_page_url are mirror opposites of the first two list items, so there is nothing new there.

The last part of the template we will look at is the go to page button:

```
<div class="pull-right for-page-button">  
  
  <button @click.prevent="getData(go_to_page)"  
         class="btn btn-default">  
  
    Go To Page:  
  
  </button>  
  
  <input v-model="go_to_page" class="number-input"></input>  
  
</div>
```

You can see we use v-model to bind the input value to go_to_page, then we hand that into the getData method, which is triggered by the @click event.

Now that we understand how the template is consuming the data, understanding the actual script that powers it will be fairly easy.

Let's look at the whole script, then we'll break it down:

```
<script>

  export default {

    mounted: function () {

      this.loadData();
    },


    data: function () {

      return {

        query: '',
        gridColumn: ['Id', 'Name', 'Created'],
        gridData: [],
        total: null,
        next_page_url: null,
        prev_page_url: null,
        last_page: null,
        current_page: null,
        pages: [],
        first_page_url: null,
        last_page_url: null,
        go_to_page: null,
        sortOrder: 1,
        sortKey: ''
      }
    },


    methods: {

      sortBy: function (key){

        this.sortKey = key;
        this.sortOrder = (this.sortOrder == 1) ? -1 : 1;
        this.getData(1);

      },
    }
  }
}
```

```
search: function(query){  
  
    this.getData(query);  
  
},  
  
loadData: function (){  
  
    $.getJSON('api/widget-data', function (data) {  
  
        this.gridData = data.data;  
        this.total = data.total;  
        this.last_page = data.last_page;  
        this.next_page_url = data.next_page_url;  
        this.prev_page_url = data.prev_page_url;  
        this.current_page = data.current_page;  
        this.first_page_url = 'api/widget-data?page=1';  
        this.last_page_url = 'api/widget-data?page=' + this.last_page;  
        this.setPageNumbers();  
  
    } ).bind(this));  
  
},  
  
setPageNumbers: function(){  
  
    for (var i = 1; i <= this.last_page; i++) {  
        this.pages.push(i);  
  
    }  
  
},  
  
getData: function (request){  
  
    let getPage;  
  
    switch (request){  
  
        case this.prev_page_url :  
    }
```

```
getPage = this.prev_page_url +
          '&column=' + this.sortKey +
          '&direction=' + this.sortOrder;

break;

case this.next_page_url :

getPage = this.next_page_url +
          '&column=' + this.sortKey +
          '&direction=' + this.sortOrder;

break;

case this.first_page_url :

getPage = this.first_page_url +
          '&column=' + this.sortKey +
          '&direction=' + this.sortOrder;

break;

case this.last_page_url :

getPage = this.last_page_url +
          '&column=' + this.sortKey +
          '&direction=' + this.sortOrder;

break;

case this.query :

getPage = 'api/widget-data?' +
          'keyword=' + this.query +
          '&column=' + this.sortKey +
          '&direction=' + this.sortOrder;

break;

case this.go_to_page :

if( this.go_to_page != '' && this.pageInRange()){
```

```
getPage = 'api/widget-data?' +
          'page=' + this.go_to_page +
          '&column=' + this.sortKey +
          '&direction=' + this.sortOrder +
          '&keyword=' + this.query;

        this.clearPageNumberInputBox();

    } else {

        alert('Please enter a valid page number');
    }

    break;

default :

    getPage = 'api/widget-data?' +
              'page=' + request +
              '&column=' + this.sortKey +
              '&direction=' + this.sortOrder +
              '&keyword=' + this.query;

    break;
}

if (this.query == '' && getPage != null){

    $.getJSON(getPage, function (data) {
        this.gridData = data.data;
        this.total = data.total;
        this.last_page = data.last_page;
        this.next_page_url = data.next_page_url;
        this.prev_page_url = data.prev_page_url;
        this.current_page = data.current_page;

    }).bind(this));
}

} else {

    if (getPage != null){

        $.getJSON(getPage, function (data) {
```

```
        this.gridData = data.data;
        this.total = data.total;
        this.last_page = data.last_page;
        this.next_page_url = (data.next_page_url == null) ?
        null : data.next_page_url + '&keyword=' +this.query;
        this.prev_page_url = (data.prev_page_url == null) ?
        null : data.prev_page_url + '&keyword=' +this.query;
        this.first_page_url =
        'api/widget-data?page=1&keyword=' + this.query;
        this.last_page_url = 'api/widget-data?page='
        + this.last_page + '&keyword=' +this.query;
        this.current_page = data.current_page;
        this.resetPageNumbers();

    }.bind(this));

}

}

,

checkPage: function(page){

    return page == this.current_page;

},

resetPageNumbers: function(){

    this.pages = [];

    for (var i = 1; i <= this.last_page; i++) {
        this.pages.push(i);

    }

}

checkUrlNotNull: function(url){

    return url != null;

}
```

```
    } ,  
  
    clearPageNumberInputBox: function(){  
  
        return this.go_to_page = '';  
  
    } ,  
  
    pageInRange: function(){  
  
        return this.go_to_page <= parseInt(this.last_page);  
  
    }  
}  
}  
  
</script>
```

Ok, let's break this down. You can see we open with:

```
export default {
```

Then comes our mounted method:

```
mounted: function () {  
  
    this.loadData();  
  
},
```

The mounted method executes after the instance has been mounted to the DOM. You can see all we have it doing is calling loadData, which is our method for loading the initial data.

Then we get our data:

```

data: function () {

    return {

        query: '',
        gridColumns: ['Id', 'Name', 'Created'],
        gridData: [],
        total: null,
        next_page_url: null,
        prev_page_url: null,
        last_page: null,
        current_page: null,
        pages: [],
        first_page_url: null,
        last_page_url: null,
        go_to_page: null,
        sortOrder: 1,
        sortKey: ''
    }
}

},

```

Note that on a component, data must be returned within a function.

You can see in the object we are returning, these are mostly empty values. Obviously the gridColumns and the sortOrder are exceptions.

Now we can look at our loadData method:

```

loadData: function (){

    $.getJSON('api/widget-data', function (data) {

        this.gridData = data.data;
        this.total = data.total;
        this.last_page = data.last_page;
        this.next_page_url = data.next_page_url;
        this.prev_page_url = data.prev_page_url;
    })
}

```

```

    this.current_page = data.current_page;
    this.first_page_url = 'api/widget-data?page=1';
    this.last_page_url = 'api/widget-data?page=' + this.last_page;
    this.setPageNumbers();

}.bind(this));

},

```

In our loadData function, we are actually using a jQuery call to send a get request.

Laravel includes the [Axios library](#), which uses promises and has a very intuitive api.

I'm currently using jQuery because I'm more used to it. Anyway, it's an example of how you can mix libraries, since we also have access to jQuery.

We consume the data back from the ajax request, and the format for this is that we return a data object, to which the json we are returning belongs. That is why we set this.gridData to data.data. Hopefully that is clear.

We are also calling setPageNumbers():

```

setPageNumbers: function(){
  for (var i = 1; i <= this.last_page; i++) {
    this.pages.push(i);
  }
},

```

What this does is fill in the pages array with the number for each page in the range.

Note at the end of the call, I'm using:

```
.bind(this));
```

That binds the result to the Vue component.

Next, let's look at the sortBy method:

```
sortBy: function (key){  
  
  this.sortKey = key;  
  this.sortOrder = (this.sortOrder == 1) ? -1 : 1;  
  this.getData(1);  
  
},
```

So this is pretty straightforward. At the end we call `this.getData(1)`, which will paginate the results from the first page of sorted results.

Since we already covered the search method and `loadData`, we are ready to look at `getData`:

```
getData: function (request){  
  
  let getPage;  
  
  switch (request){  
  
    case this.prev_page_url :  
  
      getPage = this.prev_page_url +  
        '&column=' + this.sortKey +  
        '&direction=' + this.sortOrder;  
  
      break;  
  
    case this.next_page_url :  
  
      getPage = this.next_page_url +  
        '&column=' + this.sortKey +  
        '&direction=' + this.sortOrder;  
  
      break;  
  
    case this.first_page_url :  
  
      getPage = this.first_page_url +
```

```
    '&column=' + this.sortKey +
    '&direction=' + this.sortOrder;

break;

case this.last_page_url :

    getPage = this.last_page_url +
        '&column=' + this.sortKey +
        '&direction=' + this.sortOrder;

break;

case this.query :

    getPage = 'api/widget-data?' +
        'keyword=' + this.query +
        '&column=' + this.sortKey +
        '&direction=' + this.sortOrder;

break;

case this.go_to_page :

    if( this.go_to_page != '' && this.pageInRange()) {

        getPage = 'api/widget-data?' +
            'page=' + this.go_to_page +
            '&column=' + this.sortKey +
            '&direction=' + this.sortOrder +
            '&keyword=' + this.query;

        this.clearPageNumberInputBox();

    } else {

        alert('Please enter a valid page number');

    }

break;

default :
```

```
getPage = 'api/widget-data?' +
    'page=' + request +
    '&column=' + this.sortKey +
    '&direction=' + this.sortOrder +
    '&keyword=' + this.query;

        break;
}

if (this.query == '' && getPage != null){

    $.getJSON(getPage, function (data) {

        this.gridData = data.data;
        this.total = data.total;
        this.last_page = data.last_page;
        this.next_page_url = data.next_page_url;
        this.prev_page_url = data.prev_page_url;
        this.current_page = data.current_page;

    }).bind(this));

} else {

    if (getPage != null){

        $.getJSON(getPage, function (data) {

            this.gridData = data.data;
            this.total = data.total;
            this.last_page = data.last_page;
            this.next_page_url = (data.next_page_url == null) ?
                null : data.next_page_url + '&keyword=' +this.query;
            this.prev_page_url = (data.prev_page_url == null) ?
                null : data.prev_page_url + '&keyword=' +this.query;
            this.first_page_url =
                'api/widget-data?page=1&keyword=' +this.query;
            this.last_page_url = 'api/widget-data?page=' +
                this.last_page +
                '&keyword=' +this.query;
            this.current_page = data.current_page;
            this.resetPageNumbers();

        }).bind(this));

    }
}
```

```
    }  
}  
  
},
```

Let's break this up. We start with:

```
getData: function (request){  
  
  let getPage;  
  
  switch (request){
```

So we start by taking in the request value and setting a variable named getPage. We are using ES6 syntax for let, which defines the variable within the code block only.

Then we have the monster switch statement that handles all the different ways we will request data. We don't need to go through each one, but you can look them over to get familiar with what they are.

In any event, getPage gets set. So we move on to the if block:

```
if (this.query == '' && getPage != null){  
  
  $.getJSON(getPage, function (data) {  
  
    this.gridData = data.data;  
    this.total = data.total;  
    this.last_page = data.last_page;  
    this.next_page_url = data.next_page_url;  
    this.prev_page_url = data.prev_page_url;  
    this.current_page = data.current_page;  
  
    }.bind(this));  
  
}
```

We check to see if the query has not been set, in other words, no keyword given, and if getPage is not null, we make the ajax call.

If the query variable has a value other than an empty string, we get the else block:

```
else {

  if (getPage != null){

    $.getJSON(getPage, function (data) {

      this.gridData = data.data;
      this.total = data.total;
      this.last_page = data.last_page;
      this.next_page_url = (data.next_page_url == null) ?
        null : data.next_page_url + '&keyword=' +this.query;
      this.prev_page_url = (data.prev_page_url == null) ?
        null : data.prev_page_url + '&keyword=' +this.query;
      this.first_page_url =
        'api/widget-data?page=1&keyword=' + this.query;
      this.last_page_url = 'api/widget-data?page=' + this.last_page +
        '&keyword=' +this.query;
      this.current_page = data.current_page;
      this.resetPageNumbers();

    }).bind(this));

  }
}
```

So that's how we handle the various requests from the data grid. You may have noticed:

`this.resetPageNumbers();`

Since we're not simply sorting, we are filtering by keyword, we need to reset the page numbers with the following method;

```
resetPageNumbers: function(){  
  
    this.pages = [];  
  
    for (var i = 1; i <= this.last_page; i++) {  
  
        this.pages.push(i);  
  
    }  
  
},
```

We are resetting the pages array based on the number of pages indicated the last_page property.
And that pretty much covers our WidgetGrid component.

MarketingImage Data Grid

I'm going to step you through one more implementation of the MarketingImage grid, just to show you how easy it is to create these components, once you have one to follow as an example.

So the steps will be:

- make an api route for marketing-image-data
- add a marketingImageData method to the ApiController
- make a MarketingImageQuery class to supply our data
- make a MarketingImageWidget.vue component
- add the call to the marketing-image-widget in app.js
- compile via npm run dev
- adjust the index method of the MarketingImageController
- adjust the index view of views/marketing-image

API route

Add the following route to your routes/web.php file:

```
Route::get('api/marketing-image-data', 'ApiController@marketingImageData');
```

marketingImageData method on Api Controller

Add the following method to your ApiController:

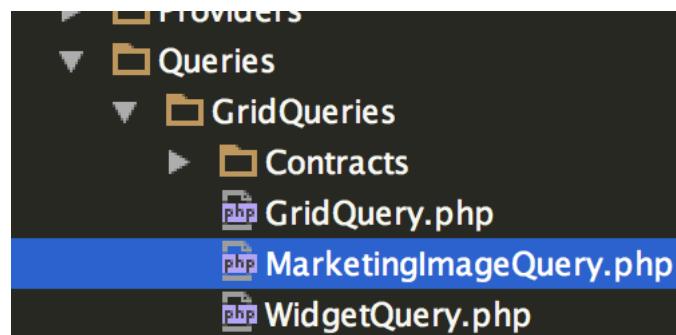
```
public function marketingImageData(Request $request)
{
    return GridQuery::sendData($request, new MarketingImageQuery);
}
```

Add the following use statement to the ApiController:

```
use App\Queries\GridQueries\MarketingImageQuery;
```

MarketingImageQuery.php

Next we'll create the MarketingImageQuery.php file here:



Let's put the following in MarketingImageQuery.php:

Gist:

[MarketingImageQuery.php](#)

From book:

```
<?php

namespace App\Queries\GridQueries;
use DB;
use App\Queries\GridQueries\Contracts\DataQuery;

class MarketingImageQuery implements DataQuery
{

    public function data($column, $direction)
    {

        $rows = DB::table('marketing_images')
            ->select('id as Id',
                      'image_name as Name',
                      'image_extension as Ext',
                      'image_weight as Weight',
                      'is_featured as Featured',
                      'is_active as Active',
                      DB::raw('DATE_FORMAT(created_at,
                            "%m-%d-%Y") as Created'))
            ->orderBy($column, $direction)
            ->paginate(10);

        return $rows;
    }

    public function filteredData($column, $direction, $keyword)
    {

        $rows = DB::table('marketing_images')
            ->select('id as Id',
                      'image_name as Name',
                      'image_extension as Ext',
                      'image_weight as Weight',
                      'is_featured as Featured',
                      'is_active as Active',
                      DB::raw('DATE_FORMAT(created_at,
                            "%m-%d-%Y") as Created'))
            ->where('image_name', 'like', '%' . $keyword . '%')
            ->orderBy($column, $direction)
            ->paginate(10);
    }
}
```

```
    return $rows;  
}  
}
```

So far, nothing new worth discussing. We have a few extra selects that we will need and that's it.

Hit the following url to make sure it is all working at this point:

<http://sample-project.com/api/marketing-image-data>

You should be getting data. If not, go back and look for the typo or omission.

MarketingImageGrid.vue

Next let's create the MarketingImageGrid.vue file. We are not going to review the whole thing, just the parts that are different, but I do need to give you the entire file:

Gist:

[MarketingImageGrid.vue](#)

From book:

```
<template>  
  
<div class="row">  
  
  <div class="col-lg-12">  
  
    <form id="search">  
  
      Search  
  
      <input name="query"  
             v-model="query"  
             @keyup="search(query)">
```

```
</form>

<div class="pull-right">
    {{ total }} Total Results
</div>

<section class="panel">
    <div class="panel-body">
        <table class="table table-bordered table-striped">
            <thead>
                <tr>
                    <th v-for="key in gridColumnss"
                        @click="sortBy(key)"
                        v-bind:class="{active: sortKey == key}">
                        {{ key }}
                    <span class="arrow"
                        v-bind:class="sortOrder > 0 ? 'asc' : 'dsc'">
                    </span>
                </th>
                <th>
                    Actions
                </th>
            </thead>
            <tbody>
                <tr v-for="row in gridData">
```

```
<td>

  <a v-bind:href="'/marketing-image/' + row.Id">
    
  </a>

</td>

<td>

  <a v-bind:href="'/widget/' + row.Id + '-' + row.Slug">
    {{ row.Name }}
  </a>

</td>

<td>
  {{ row.Weight }}
</td>

<td>

  {{ convertBoolean(row.Featured) }}
</td>

<td>

  {{ convertBoolean(row.Active) }}
</td>

<td>

  {{ row.Created }}
</td>

<td>
```

```
<a v-bind:href="'/widget/' + row.Id + '/edit'">

    <button type="button"
        class="btn btn-default">

        Edit

    </button>

</a>
</td>

</tr>

</tbody>

</table>
</div>

<div class="pull-right">

    page {{ current_page }} of {{ last_page }} pages

</div>

</section>

<div class="row">

    <div class="pull-right for-page-button">

        <button @click.prevent="getData(go_to_page)"
            class="btn btn-default">

            Go To Page:

        </button>

        <input v-model="go_to_page" class="number-input"></input>

    </div>

    <ul class="pagination pull-right">
```

```
<li>

  <a @click.prevent="getData(first_page_url)">
    first
  </a>

</li>

<li v-if="checkUrlNotNull(prev_page_url)">
  <a @click.prevent="getData(prev_page_url)">
    prev
  </a>

</li>

<li v-for="page in pages"
  v-if="page > current_page - 2 && page < current_page + 2"
  v-bind:class="{ 'active': checkPage(page) }">
  <a @click.prevent="getData(page)">
    {{ page }}
  </a>
</li>

<li v-if="checkUrlNotNull(next_page_url)">
  <a @click.prevent="getData(next_page_url)">
    next
  </a>
</li>

<li>
```

```
<a @click.prevent="getData(last_page_url)">  
    last  
</a>  
</li>  
</ul>  
</div>  
</div>  
</div>  
</template>  
  
<script>  
export default {  
  
  mounted: function () {  
  
    this.loadData();  
  },  
  
  data: function () {  
  
    return {  
  
      query: '',  
      gridColumns: ['Thumbnail', 'Name', 'Weight',  
                    'Featured', 'Active', 'Created'],  
      gridData: [],  
      total: null,  
      next_page_url: null,  
      prev_page_url: null,  
      last_page: null,  
      current_page: null,  
      pages: [],  
      first_page_url: null,  
    };  
  },  
};
```

```
        last_page_url: null,
        go_to_page: null,
        sortOrder: 1,
        sortKey: ''
    }

},
methods: {
    sortBy: function (key){
        this.sortKey = key;
        this.sortOrder = (this.sortOrder == 1) ? -1 : 1;
        this.getData(1);
    },
    search: function(query){
        this.getData(query);
    },
    loadData: function (){
        $.getJSON('api/marketing-image-data', function (data) {
            this.gridData = data.data;
            this.total = data.total;
            this.last_page = data.last_page;
            this.next_page_url = data.next_page_url;
            this.prev_page_url = data.prev_page_url;
            this.current_page = data.current_page;
            this.first_page_url = 'api/marketing-image-data?page=1';
            this.last_page_url = 'api/marketing-image-data?page='
                + this.last_page;
            this.setPageNumbers();
        }.bind(this));
    }
}
```

```
},  
  
setPageNumbers: function(){  
  
    for (var i = 1; i <= this.last_page; i++) {  
  
        this.pages.push(i);  
  
    }  
  
},  
  
convertBoolean: function (value){  
  
    return value == 1 ? 'Yes' : 'No';  
  
},  
  
getData: function (request){  
  
    let getPage;  
  
    switch (request){  
  
        case this.prev_page_url :  
  
            getPage = this.prev_page_url +  
                '&column=' + this.sortKey +  
                '&direction=' + this.sortOrder;  
  
            break;  
  
        case this.next_page_url :  
  
            getPage = this.next_page_url +  
                '&column=' + this.sortKey +  
                '&direction=' + this.sortOrder;  
  
            break;  
  
        case this.first_page_url :  
    }
```

```
getPage = this.first_page_url +
    '&column=' + this.sortKey +
    '&direction=' + this.sortOrder;

break;

case this.last_page_url :

getPage = this.last_page_url +
    '&column=' + this.sortKey +
    '&direction=' + this.sortOrder;

break;

case this.query :

getPage = 'api/marketing-image-data?' +
    'keyword=' + this.query +
    '&column=' + this.sortKey +
    '&direction=' + this.sortOrder;

break;

case this.go_to_page :

if( this.go_to_page != '' && this.pageInRange()){

getPage = 'api/marketing-image-data?' +
    'page=' + this.go_to_page +
    '&column=' + this.sortKey +
    '&direction=' + this.sortOrder +
    '&keyword=' + this.query;

this.clearPageNumberInputBox();

} else {

    alert('Please enter a valid page number');
}

break;

default :
```

```
getPage = 'api/marketing-image-data?' +
    'page=' + request +
    '&column=' + this.sortKey +
    '&direction=' + this.sortOrder +
    '&keyword=' + this.query;

        break;
}

if (this.query == '' && getPage != null){

    $.getJSON(getPage, function (data) {

        this.gridData = data.data;
        this.total = data.total;
        this.last_page = data.last_page;
        this.next_page_url = data.next_page_url;
        this.prev_page_url = data.prev_page_url;
        this.current_page = data.current_page;

    }).bind(this));

} else {

    if (getPage != null){

        $.getJSON(getPage, function (data) {

            this.gridData = data.data;
            this.total = data.total;
            this.last_page = data.last_page;
            this.next_page_url = (data.next_page_url == null) ?
                null : data.next_page_url + '&keyword=' +this.query;
            this.prev_page_url = (data.prev_page_url == null) ?
                null : data.prev_page_url + '&keyword=' +this.query;
            this.first_page_url =
                'api/marketing-image-data?page=1&keyword='
                +this.query;
            this.last_page_url = 'api/marketing-image-data?page='
                + this.last_page + '&keyword=' +this.query;
            this.current_page = data.current_page;
            this.resetPageNumbers();

        });

    }

}
```

```
        } .bind(this));

    }

} ,

checkPage: function(page){

    return page == this.current_page;

} ,

resetPageNumbers: function(){

    this.pages = [];

    for (var i = 1; i <= this.last_page; i++) {

        this.pages.push(i);

    }

} ,

checkUrlNotNull: function(url){

    return url != null;

} ,

clearPageNumberInputBox: function(){

    return this.go_to_page = '';

} ,

pageInRange: function(){

    return this.go_to_page <= parseInt(this.last_page);

}
```

```
        }  
    }  
  
</script>
```

Definitely use the Gist for code format.

So this is mostly similar to what we did before. Let's look at the differences in the template first. Since the table headings are generated dynamically, we don't need to change anything there. But we do have changes in the table body:

```
<tbody>  
  
<tr v-for="row in gridData">  
  
    <td>  
  
        <a v-bind:href="'/marketing-image/' + row.Id">  
              
        </a>  
  
    </td>  
  
    <td>  
  
        <a v-bind:href="'/widget/' + row.Id + '-' + row.Slug">  
            {{ row.Name }}  
        </a>  
  
    </td>  
  
    <td>  
        {{ row.Weight }}  
    </td>
```

```
<td>

    {{ convertBoolean(row.Featured) }}

</td>

<td>

    {{ convertBoolean(row.Active) }}

</td>

<td>

    {{ row.Created }}

</td>

<td >

    <a v-bind:href="/widget/' + row.Id + '/edit'">

        <button type="button"
                class="btn btn-default">

            Edit

        </button>

    </a>
</td>

</tr>

</tbody>
```

The first `<td>` is different because now we are pulling a thumbnail:

```
<td>

  <a v-bind:href="'/marketing-image/' + row.Id">
    
  </a>

</td>
```

Obviously, we are linking it to the show record like we did before. Note we are being explicit about the image path, without using a helper. It's up to you if you want to create a helper for it in javascript, but I felt it was unnecessary in this case.

Ok, next we have:

```
<td>

  <a v-bind:href="'/marketing-image/' + row.Id">
    {{ row.Name }}
  </a>

</td>
```

That just makes a link to the show record from the Name field.

We can do the next two together, since they are the same:

```
<td>  
    {{ convertBoolean(row.Featured) }}  
</td>  
  
<td>  
    {{ convertBoolean(row.Active) }}  
</td>
```

We made a convertBoolean method to change the 1 and 0 values to display ‘Yes’ and ‘No.’

Ok, let’s look at that method on the script:

```
convertBoolean: function (value){  
  
    return value == 1 ? 'Yes' : 'No';  
  
},
```

That is fairly straightforward, it just takes in the value and check to see if equals one, and if so, return ‘Yes’, if not, then return ‘No.’

So just two changes left. First let’s look at the gridColumns array in data:

```
gridColumns: ['Thumbnail', 'Name', 'Weight', 'Featured', 'Active', 'Created'],
```

And the only other change we have is to go around put in the right path for the ajax call in loadData and getData methods:

```
api/marketing-image-data
```

There are a number of spots for that, so if are copying an existing Vue file to build off of, then make sure you get them all.

And that’s it for MarketingImageGrid.vue file.

Now we need to add a call to it from resources/assets/js/app.js:

```
Vue.component('marketing-image-grid',  
  require('./components/MarketingImageGrid.vue'));
```

That's one line in my IDE.

npm run dev

Now we run npm run dev from the command line:

```
npm run dev
```

MarketingImageController.php

Next we need to modify the index method of the MarketingImageController to the following:

```
public function index()  
{  
  
    return view('marketing-image.index');  
  
}
```

marketing-image/index.blade.php

And finally, we need to change resources/views/marketing-image/index.blade.php to the following:

Gist:

[index.blade.php](#)

From book:

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

@section('title')
    <title>Marketing Images</title>

@endsection

@section('content')
    <ol class='breadcrumb'>
        <li><a href='/'>Home</a></li>
        <li class='active'>Marketing Images</li>
    </ol>

    <h2>Marketing Images</h2>

    <hr/>

    <marketing-image-grid></marketing-image-grid>

    <div> <a href="/marketing-image/create">
        <button type="button" class="btn btn-lg btn-primary">
            Create New
        </button>
    </a>
    </div>

    @endsection
```

And that should all be working for you now, but... There's a problem.

Remember when we set the default order on the marketing images to show the one with the most weight first? Well, currently, we don't have a way to control that.

So we need to make a simple change to GirdQuery.php. I'm going to give you the whole file, then we can discuss:

Gist:

[GridQuery.php](#)

From book:

```
<?php

namespace App\Queries\GridQueries;

use Illuminate\Http\Request;
use App\Queries\GridQueries\Contracts\DataQuery;

class GridQuery
{

    public static function sendData(Request $request, DataQuery $query)
    {

        // set sort by column and direction

        list($column, $direction) = static::setSort($request, $query);

        // search by keyword with column sort

        if ($request->has('keyword')) {

            return static::keywordFilter($request, $query, $column, $direction);

        }

        // return data

        return static::getData($query, $column, $direction);

    }

    public static function setSort(Request $request, $query)
    {
```

```
// set sort by column with default

list($column, $direction) = static::setDefaults($query);

if ($request->has('column')) {

    $column = $request->get('column');

    if ($column == 'Id') {

        $direction = $request->get('direction') == 1 ? 'asc' : 'desc';

        return [$column, $direction];

    } else {

        $direction = $request->get('direction') == 1 ? 'desc' : 'asc';

        return [$column, $direction];

    }

}

return [$column, $direction];

}

public static function setDefaults($query)
{

switch ($query){

    case $query instanceof MarketingImageQuery :

        $column = 'image_weight';
        $direction = 'asc';

        break;

    default:
```

```
$column = 'id';
$direction = 'asc';

break;

}

return list($column, $direction) = [$column, $direction];

}

public static function keywordFilter
    (Request $request, $query, $column, $direction)
{
    $keyword = $request->get('keyword');

    return response()->json($query->filteredData($column,
                                                    $direction,
                                                    $keyword));
}

public static function getData($query, $column, $direction)
{
    return response()->json($query->data($column, $direction));
}

}
```

So in the setSort method, we now have the following call:

```
list($column, $direction) = static::setDefaults($query);
```

That calls the following method:

```
public static function setDefaults($query)
{
    switch ($query){
        case $query instanceof MarketingImageQuery :

            $column = 'image_weight';
            $direction = 'asc';

            break;

        default:

            $column = 'id';
            $direction = 'asc';

            break;
    }

    return list($column, $direction) = [$column, $direction];
}
```

So we have a switch statement that defaults to what we had before:

```
default:

$column = 'id';
$direction = 'asc';

break;
```

And above that, we check to see if the query handed in is an instance of MarketingImageQuery:

```
case $query instanceof MarketingImageQuery :

    $column = 'image_weight';
    $direction = 'asc';

    break;
```

If you make another query class that has special defaults, just add another case statement for it.

So you can see how we are handling the special cases, where we need something other than the default.

Let's check it out, it should put the image with the lowest weight number first, like so:

| Thumbnail | Name | Weight | Featured | Active | Created | Actions |
|-----------|------------|--------|----------|--------|------------|-----------------------|
| | Screenshot | 40 | No | Yes | 01-28-2017 | <button>Edit</button> |
| | laravel | 90 | No | Yes | 01-28-2017 | <button>Edit</button> |
| | Wonder | 100 | Yes | Yes | 01-28-2017 | <button>Edit</button> |

page 1 of 1 pages

first **1** last Go To Page:

[Create New](#)

© 2015 - 2017 Sample Project All rights Reserved.

Queries can be incredibly complex, and you can easily outgrow your capacity to manage it from our GridQuery implementation, but that is for you to decide. This implementation gets you up and running and will work fine for simpler queries like those that we have used.

You can use the knowledge you have now to build your User and Profile components if you wish to use Vue.js for that.

Summary

Building out an efficient way to create queries for our project will have lasting benefits. You can build upon and customize these classes as you see fit.

You can see that reducing the ApiController method down to a single line call makes for a very clean controller, and that makes it extensible, maximum separation for reusability.

So that wraps up our basic book. I would really appreciate it if you could leave a positive review or rating at [GoodReads](#).

I will no doubt add some bonus material at some point, so keep an eye out for that. Thanks again for taking the Laravel journey with me. See you soon.

Bill.

Chapter 13: Events, Mail, and Architecture.

Mail

Laravel has made creating custom email messages easy and simple. Our application is already configured for mailtrap.io. If you have not set it up for that, please do so now. You can get a free account from [mailtrap.io](#) and sign up with a single click from Github.

Once you have that, plugin your username and password, both are long strings of numbers and letters, not to be confused with your login information:

```
MAIL_DRIVER="smtp"
MAIL_HOST="mailtrap.io"
MAIL_PORT=2525
MAIL_USERNAME=your-username
MAIL_PASSWORD=your-password
MAIL_ENCRYPTION=null
```

You should be all set, once that's in place.

When you actually go into production, you should check config/mail.php to see which services are supported out of the box. As of this writing they include:

- mailgun
- mandrill
- sparkpost

You would then change your .env to reflect the service you wanted.

Email Confirmation On Registration

The next thing we are going to do is create an email that gets sent to users upon successful registration. I think you'll be amazed at how simple it really is.

Let's copy the RegistersUsers register method and place it into our AuthController.

Gist:

[AuthController.php](#)

From book:

```
public function register(Request $request)
{
    $this->validator($request->all())->validate();

    event(new Registered($user = $this->create($request->all())));

    $this->guard()->login($user);

    return $this->registered($request, $user) ? 
        redirect($this->redirectTo());
}

}
```

In the Gist, I provided the full AuthController, which includes the following use statement:

```
use Illuminate\Auth\Events\Registered;
```

So we're going to overwrite the trait method by including it on the AuthController. This means we can add functionality, while causing minimal change to the original trait. This makes it easier to maintain with future version releases.

Inside the create method, we will add a line before the return statement:

```

public function register(Request $request)
{
    $this->validator($request->all())->validate();

    event(new Registered($user = $this->create($request->all())));

    \Mail::to($user)->send(new RegistrationEmail)

    $this->guard()->login($user);

    return $this->registered($request, $user) ?:
        redirect($this->redirectTo());
}


```

So we added the line:

```
\Mail::to($user)->send(new RegistrationEmail);
```

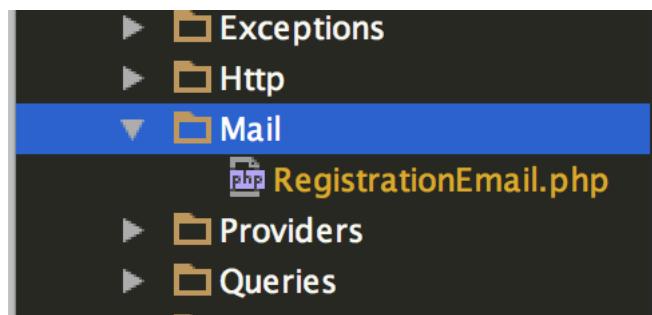
Without going into it very deeply, we can see how intuitive the syntax is. We call the Mail Class's static methods, to and send. We are telling it send a new RegistrationEmail, but we have not created that yet.

You can name this email anything you want. You could call it WelcomeEmail. I like RegistrationEmail because it reminds me they get it on registration.

We will create the new class by using the following artisan command:

```
php artisan make:mail RegistrationEmail
```

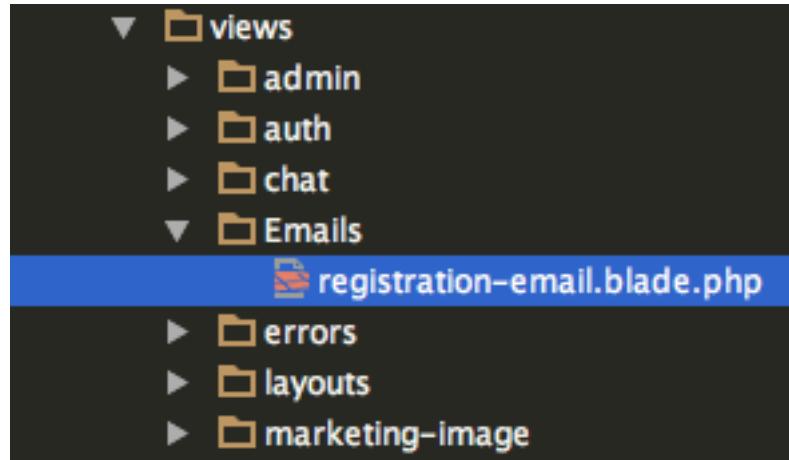
After running this, you will see a new mail folder in your app directory:



Now that we have this file, let's add the use statement to the top of the AuthController:

```
use App\Mail\RegistrationEmail;
```

Next, let's create view folder for our emails, we'll name the folder emails:



Let's create registration-email.blade.php with the following contents:

Gist:

[registration-email.blade.php](#)

From book:

```
<!DOCTYPE html>

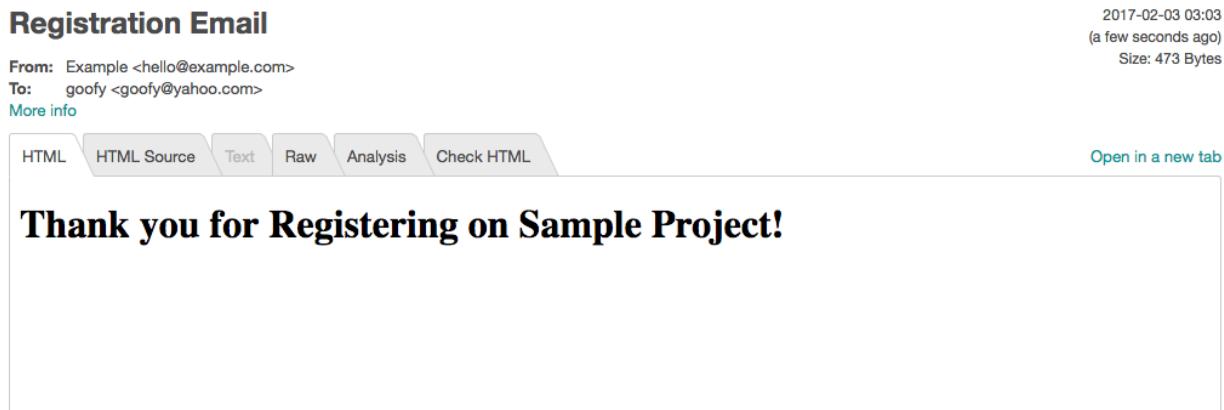
<html>
  <head>
    <title>Sample Project Registration</title>
  </head>
  <body>
    <h1> Thank you for Registering on Sample Project! </h1>
  </body>
```

```
</html>
```

Now we can modify RegistrationEmail.php to point to the right view in the build method:

```
public function build()
{
    return $this->view('emails.registration-email');
}
```

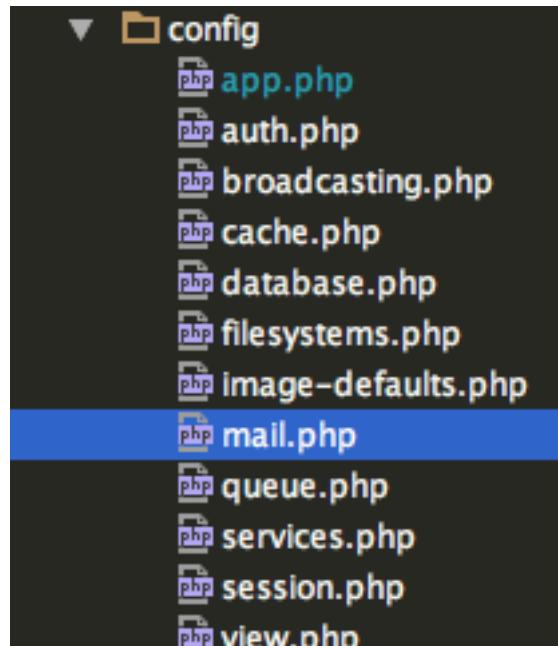
At this point we have everything in place to make this work, so let's give it a try by registering a new user. Once that's done, check your mailtrap.io inbox:



Note the from address is not what we want.

config/mail.php

Let's go to config/mail.php:



Let's find the following in the file:

```
'from' => [
    'address' => env('MAIL_FROM_ADDRESS', 'hello@example.com'),
    'name' => env('MAIL_FROM_NAME', 'Example'),
],
```

Let's change it:

Gist:

[from array](#)

From book:

```
'from' => [
    'address' => env('MAIL_FROM_ADDRESS',
        'support@sample-project.com'),
    'name' => env('MAIL_FROM_NAME', 'Support Team'),
],
]
```

You can register another user to check that out.

Passing Data to the Email

Next we'll pass dynamic information through to the email, in this case the username. We're not limited to that, but this will suffice to demonstrate.

We easily pass in the \$user, since we already have that in our register method on our AuthController.

Let's change the Mail call to the following:

```
\Mail::to($user)->send(new RegistrationEmail($user));
```

You can see we added \$user to the constructor of RegistrationEmail. So let's modify the constructor on that class and add the property:

Gist:

[RegisterEmail.php](#)

From book:

```
public $user;

/**
 * Create a new message instance.
 *
 * @return void
 */

public function __construct(User $user)
{
```

```
$this->user = $user;  
}
```

We bring in an instance of the user through the constructor and set it to a public property named \$user. The public properties are available to the view.

We will also have to add the use statement for the user:

```
use App\User;
```

Next, let's do a slight modification to our views/emails/registration-email.blade.php file:

Gist:

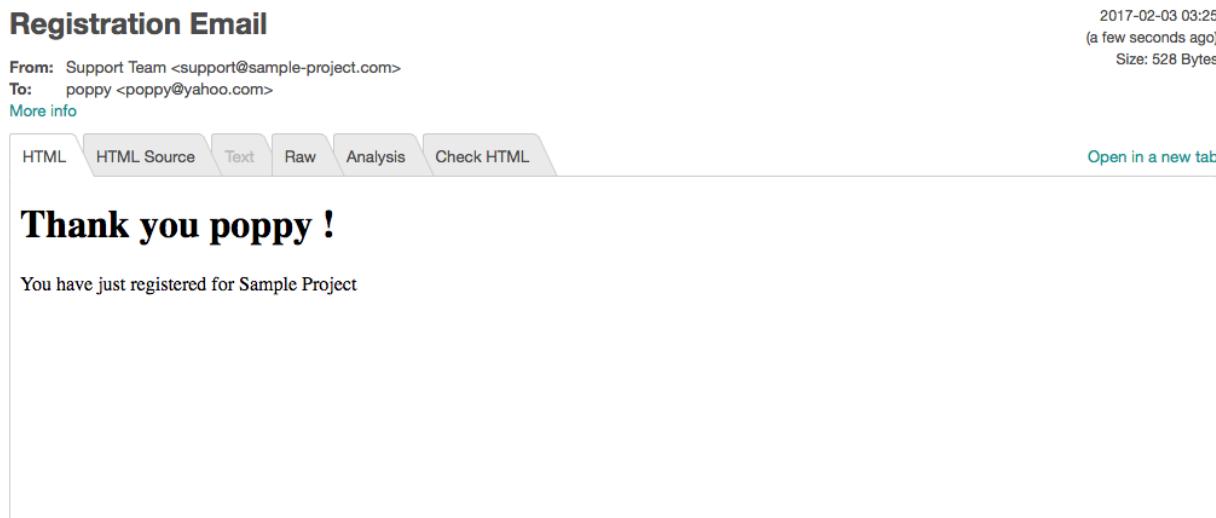
[registration-email.blade.php](#)

From book:

```
<!DOCTYPE html>  
  
<html>  
  
<head>  
  
<title>Sample Project Registration</title>  
  
</head>  
  
<body>  
  
<h1> Thank you {{ $user->name }}! </h1>  
  
<div>  
  
<p>You have just registered for Sample Project</p>  
  
</div>  
  
</body>
```

```
</html>
```

If you register another user, in my case I named the user “poppy,” the email in your inbox will look something like this:



Super simple, is it not?

Markdown Email

Laravel 5.4 also provides help to developers who want nicer formatting for their emails by supporting markdown. We'll do an example of this by first deleting `Mail/RegistrationEmail.php` and `registration-email.blade.php`.

Then we'll go to the command line, and this time we will add a flag when we run the `make:mail` command:

```
php artisan make:mail RegistrationEmail --markdown="emails.registration-email"
```

That gets us the following:

```
<?php

namespace App\Mail;

use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;
use Illuminate\Contracts\Queue\ShouldQueue;

class RegistrationEmail extends Mailable
{
    use Queueable, SerializesModels;

    /**
     * Create a new message instance.
     *
     * @return void
     */

    public function __construct()
    {
        //
    }

    /**
     * Build the message.
     *
     * @return $this
     */

    public function build()
    {

        return $this->markdown('emails.registration-email');
    }
}
```

You can see the change in the build method:

```
public function build()
{
    return $this->markdown('emails.registration-email');
}
```

We need to add in the \$user to the constructor, set the property, and add the use statement like we did before:

```
<?php

namespace App\Mail;

use App\User;
use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;
use Illuminate\Contracts\Queue\ShouldQueue;

class RegistrationEmail extends Mailable
{
    use Queueable, SerializesModels;

    public $user;

    /**
     * Create a new message instance.
     *
     * @return void
     */

    public function __construct(User $user)
    {
        $this->user = $user;
    }
}
```

If you go to your emails folder, you will see that our artisan command created the registration-email.blade.php file for us:

```
@component('mail::message')

# Introduction

The body of your message.

@component('mail::button', ['url' => ''])

Button Text

@endcomponent

Thanks, <br>

{{ config('app.name') }}
```

So what you are seeing here is the @component directive in blade, which is new in Laravel 5.4. In general, I haven't worked much with the components in blade, I don't find it as intuitive as using directives such as @extends and @section.

You can read up further on blade components in the [docs](#).

Anyway, a component goes into a slot into a master page, so the parts that we define between the components here will be assembled into the full email.

We can modify our view like so:

Gist:

[registration-email.blade.php](#)

From book:

```
@component('mail::message')
```

```
# Registration Confirmation
```

```
## Congratulations {{ $user->name }}!
```

You have joined our site **and** now have access to the following benefits:

- * instant access
- * 24/7 support
- * updated daily content

```
@component('mail::button', ['url' => 'http://www.sample-project.com'])
```

Visit Now

```
@endcomponent
```

Thanks,

```
{{ config('app.name') }}
```

```
@component('mail::panel', ['url' => ''])
```

You are receiving **this** email because you subscribed to Sample Project.
You may Unsubscribe by clicking [here](/unsubscribe).

```
@endcomponent
```

```
@endcomponent
```

We just did a little formatting with markdown. If you are not familiar with markdown, you can learn it in about 10 minutes with this free [online markdown tutorial](#)

Anyway, let's do another test registration. If you check your inbox, you will see something like the following:

Sample Project

Registration Confirmation

Congratulations froggy!

You have joined our site and now have access to the following benefits:

- instant access
- 24/7 support
- updated daily content

[Visit Now](#)

Thanks,
Sample Project

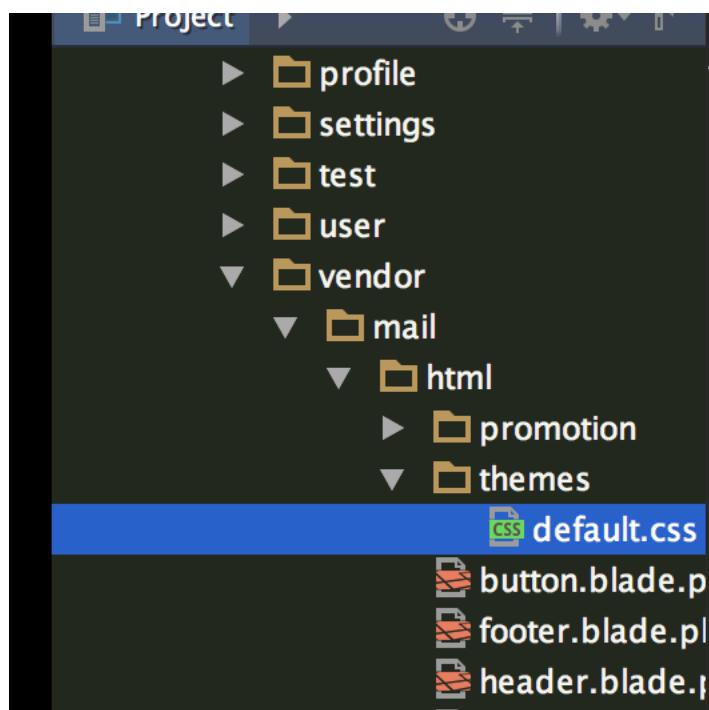
You are receiving this email because you subscribed to Sample Project. You may Unsubscribe by clicking [here](#).

© 2017 Sample Project. All rights reserved.

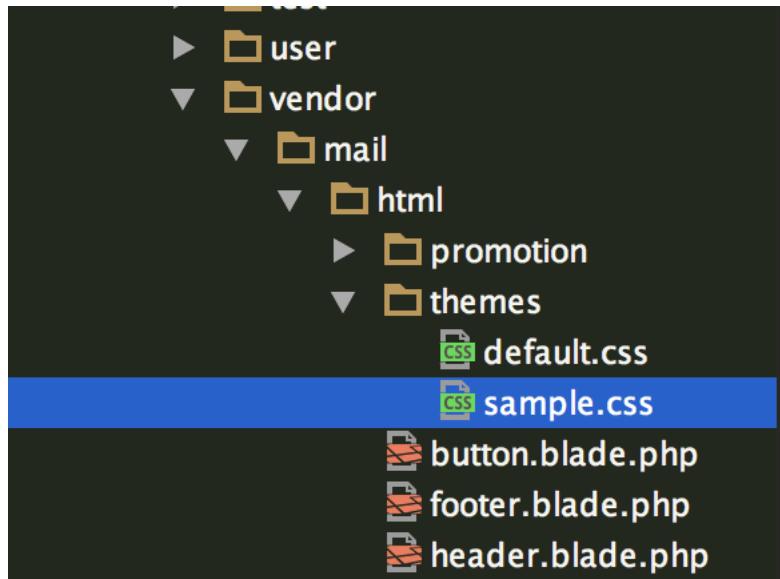
Not bad, and it was so simple to put together. As a person who is more of a backend developer, I really appreciate the help that Laravel is giving us in creating a nicely formatted email.

Custom Themes

If you want to customize the css for your theme, this is really easy to do. In a previous chapter, we published the vendor assets, so we already have a views/vendor/mail folder. Within the mail folder is a themes folder, where you will find default.css:



Let's make a copy of that file in the same folder and name it sample.css:



Now, if you want to change the css, you can make changes in the sample.css file. One last thing we need to do to make that work is to change the config/mail.php setting. This is what it is set to now:

```
'markdown' => [  
    'theme' => 'default',
```

We need to change it to:

```
'markdown' => [  
    'theme' => 'sample',
```

And now it will pull from sample.css instead of default.css.

Events

Events are a way to broadcast to your application that something has happened. There are typically 3 elements to creating an event:

- create the event class
- create the listener class
- register the event and the listener in the EventServiceProvider class

This sounds like a lot of work, but Laravel makes this almost effortless. Let's step through this and you will see what I mean.

The syntax for sending a mail event is so intuitive, we don't need to wrap it in an event. That said, you could wrap it in an event, so we will do that for demonstration purposes.

Registering the Event

So the first step will be to register the event and the listener in Providers/EventServiceProvider.php. Let's open the EventServiceProvider.php file and see what we get out of the box:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Event;
use Illuminate\Foundation\Support\Providers\EventServiceProvider
as ServiceProvider;

class EventServiceProvider extends ServiceProvider
{
    /**
     * The event listener mappings for the application.
     *
     * @var array
     */

    protected $listen = [
        'App\Events\SomeEvent' => [
            'App\Listeners\EventListener',
        ],
    ];

    /**
     * Register any events for your application.
     *
     * @return void
     */

    public function boot()
    {
        parent::boot();

        //
    }
}
```

We have a listen property that is populated with a dummy key/value pair:

```
protected $listen = [
    'App\Events\SomeEvent' => [
        'App\Listeners\EventListener',
    ],
];
```

Naming Events

We are going to replace ‘SomeEvent’ and ‘EventListener’ with our own names. So the first thing we need to do is figure out what we want to name them.

You can see in the AuthController’s register method, we already have a Registered event:

```
event(new Registered($user = $this->create($request->all())));
```

So we don’t want to confuse it with that. Why don’t we call it RegistrationCompleted? And for the listener, we will call that SendRegistrationEmail. So let’s update our \$listen property in our EventServiceProvider:

```
protected $listen = [
    'App\Events\RegistrationCompleted' => [
        'App\Listeners\SendRegistrationEmail',
    ],
];
```

This tells our application that when the RegistrationCompleted event is fired, the SendRegistrationEmail listener will listen for it and handle it.

You may have noticed that the listener is in an array, and that’s because one event can be listened to by multiple listeners, for example:

```
protected $listen = [
    'App\Events\RegistrationCompleted' => [
        'App\Listeners\SendRegistrationEmail',
        'App\Listeners\LogUserIpAddress',
    ],
];
```

In this case, we would have two classes listening to the RegistrationCompleted event, one to send the registration email and the other to log the user's ip address. We're not actually going to do the LogUserIpAddress listener, so let's make sure the \$listen property is as follows:

```
protected $listen = [
    'App\Events\RegistrationCompleted' => [
        'App\Listeners\SendRegistrationEmail',
    ],
];
```

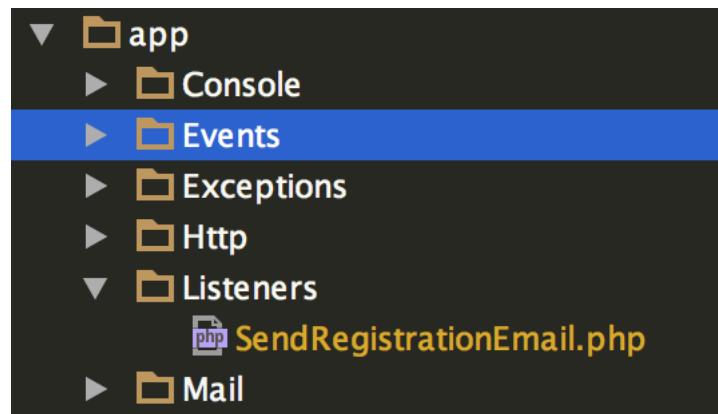
event:generate

Now we're ready to use artisan to generate these classes for us by running from the command line:

```
php artisan event:generate
```

Two new folders will be generated with the corresponding files.

Laravel makes it very convenient to use the code generators because it puts the correct folder structure in place for us and puts the right file in the right folder. :



You can see both the Events and Listeners folders have been created and you will find the appropriate file is in each folder.

Let's look at our Events/RegistrationCompleted.php file first. Here is what you get out of the box:

```
<?php

namespace App\Events;

use Illuminate\Broadcasting\Channel;
use Illuminate\Queue\SerializesModels;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Foundation\Events\Dispatchable;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;

class RegistrationCompleted
{
    use Dispatchable, InteractsWithSockets, SerializesModels;

    /**
     * Create a new event instance.
     *
     * @return void
     */

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

```

}

/**
 * Get the channels the event should broadcast on.
 *
 * @return Channel|array
 */

public function broadcastOn()
{
    return new PrivateChannel('channel-name');
}

```

We get a constructor and a broadcastOn method. The broadcastOn method is used for broadcast events like in a chat room, which we will see in action in the next chapter. We are going to leave it an empty stub here.

We can use the constructor to take in the user object, which will we need in order to process the email. Let's change the file to the following:

Gist:

[RegistrationCompleted.php](#)

From book:

```

<?php

namespace App\Events;

use App\User;
use Illuminate\Broadcasting\Channel;
use Illuminate\Queue\SerializesModels;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Foundation\Events\Dispatchable;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;

```

```
class RegistrationCompleted
{
    use Dispatchable, InteractsWithSockets, SerializesModels;

    public $user;
    /**
     * Create a new event instance.
     *
     * @return void
     */

    public function __construct(User $user)
    {
        $this->user = $user;
    }

    /**
     * Get the channels the event should broadcast on.
     *
     * @return Channel|array
     */
    public function broadcastOn()
    {
        return new PrivateChannel('channel-name');
    }
}
```

The public properties of an event are sent to the listener. Let's take a look at app/Listeners/SendRegistrationEmail.php. Here's what we get out of the box:

```
<?php

namespace App\Listeners;

use App\Events\RegistrationCompleted;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendRegistrationEmail
{
    /**
     * Create the event listener.
     *
     * @return void
     */

    public function __construct()
    {
        //
    }

    /**
     * Handle the event.
     *
     * @param RegistrationCompleted $event
     * @return void
     */

    public function handle(RegistrationCompleted $event)
    {
        //
    }
}
```

Let's go ahead and change that to:

Gist:

[SendRegistrationEmail.php](#)

From book:

```
<?php

namespace App\Listeners;

use App\Events\RegistrationCompleted;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Support\Facades\Mail;
use App\Mail\RegistrationEmail;

class SendRegistrationEmail implements ShouldQueue
{
    /**
     * Create the event listener.
     *
     * @return void
     */

    public function __construct()
    {
        //
    }

    /**
     * Handle the event.
     *
     * @param RegistrationCompleted $event
     * @return void
     */
    public function handle(RegistrationCompleted $event)
    {
        Mail::to($event->user)->send(new RegistrationEmail($event->user));
    }
}
```

So we added a couple of use statements, no surprise there:

```
use Illuminate\Support\Facades\Mail;
use App\Mail\RegistrationEmail;
```

ShouldQueue

Next we add implements ShouldQueue:

```
class SendRegistrationEmail implements ShouldQueue
```

What this does is allow us to run the event asynchronously, so our users can continue to use the application while the event is handled in the background. That's a great feature and really easy to implement as you can see.

Next, let's look at the handle method:

```
public function handle(RegistrationCompleted $event)
{
    Mail::to($event->user)->send(new RegistrationEmail($event->user));
}
```

All we are doing is calling the mail class that we created previously. Note that we are handing in \$event->user because user is a nested object brought in by \$event.

One last piece of the puzzle is to fire the event from inside the register method on the AuthController. I will supply the full controller in the gist:

Gist:

[register method](#)

From book:

```
public function register(Request $request)
{
    $this->validator($request->all())->validate();

    event(new Registered($user = $this->create($request->all())));

    $this->guard()->login($user);

    event(new RegistrationCompleted($user));

    return $this->registered($request, $user) ?:
        redirect($this->redirectTo());
}
```

Don't forget to add the use statement at the top of the file:

```
use App\Events\RegistrationCompleted;
```

So you can see we now have our event in place:

```
event(new RegistrationCompleted($user));
```

You should do a test registration to make sure it's working. And that's a basic eventing in Laravel. We will see another example, this time a broadcast event, in the next chapter.

Application Structure

The architecture of Laravel is beautiful and there are reasons why programmers love it so much. A lot of care and attention to detail was made to make the syntax helpful and clear, and the application structure makes doing complex implementations easier than they would otherwise be. Sometimes I'm amazed at how much forethought must have gone into the framework for it to be so good.

Actually, there's a 69 page book named [Laravel: From Apprentice To Artisan](#) by Taylor Otwell himself that came out in 2013. In it he lays out his vision of loose coupling, coding to a contract, and moving away from the traditional MVC concept of a model.

I'm not a fan of paying \$29.99 for 69 pages, but on the other hand, I'm so grateful for the work that Taylor has done, it seems like the least I can do. I would encourage others to do the same.

In 2013, when I read the book, I literally had no idea what he was talking about. I thought it would be a book about how to create a really good dynamic form or something along those lines. At that point in my coding development, I didn't know anything about the power of design patterns and how they play out at scale.

As we've gone through this book, you've seen many different folders, events, listeners, middleware, etc., that at first seem a bit overwhelming and overly complex. Wouldn't it be simpler to throw everything and the kitchen sink into the models? No, it would not. As it turns out, the complexity introduced by the application structure actually makes our lives simpler.

And how great is it that our command line tool, artisan, always knows where to put the specialized classes it creates for us? Remember how it created the mail folder and the appropriately namespaced file for us? I don't know about you, but I really love that.

Now that you have experienced this for yourself, let's look at the application structure itself. Don't worry about memorization, it's more important for you to understand the overall concepts. We're just taking a tour of some of fundamental aspects of the framework.

The Service Container

Your laravel application is an object instance represented as \$app, to which other classes, represented as services are bound, so they can be used by the application. That's my simple non-technical way of describing it and it still sounds fairly technical.

The \$app object is also referred to as the service container, and in the comments, you might see IoC container mentioned, which refers to Inversion of Control. If you google Inversion of control, you get:

"inversion of control (IoC) is a programming technique, expressed here in terms of object-oriented programming, in which object coupling is bound at run time by an assembler object and is typically not known at compile time using static analysis."

Again, it sounds pretty technical. Taylor Otwell designed the architecture of Laravel following formal design patterns and principles in programming.

For a formal description of the container, you can check the [container docs](#), they were written by Taylor himself. It looks like he moved away from the IoC jargon to simply call it a service container, which is a little more intuitive to what it actually does.

I have to say up front, I'm not an expert in this area and I'm nowhere near that level of programmer that could be called an architect. My analysis is not so formal and I'm not a fan of the jargon. A wise man once said the key to not drowning is not going into water that is over your head...

Anyway, bear with me, I think I can give you a working knowledge of what is happening, which is all we are really trying to accomplish here.

We'll move away from the jargon. Instead, let's take a peek under the hood and see if we can get an idea of how it all works.

When a request goes to the server, it is sent to the public folder and to the index.php file in there. The code is heavily comment with clear descriptions of what is happening:

```
<?php

/**
 * Laravel - A PHP Framework For Web Artisans
 *
 * @package Laravel
 * @author Taylor Otwell <taylor@laravel.com>
 */

/*
|--------------------------------------------------------------------------
| Register The Auto Loader
|--------------------------------------------------------------------------
|
| Composer provides a convenient, automatically generated class loader for
| our application. We just need to utilize it! We'll simply require it
| into the script here so that we don't have to worry about manual
| loading any of our classes later on. It feels nice to relax.
|
*/
require __DIR__ . '/../bootstrap/autoload.php';

/*
|--------------------------------------------------------------------------
| Turn On The Lights
|--------------------------------------------------------------------------
|
| We need to illuminate PHP development, so let us turn on the lights.
| This bootstraps the framework and gets it ready for use, then it
```

```
| will load up this application so that we can run it and send
| the responses back to the browser and delight our users.
|
*/
$app = require_once __DIR__ . '/../bootstrap/app.php';

/*
-----
| Run The Application
-----
|
| Once we have the application, we can handle the incoming request
| through the kernel, and send the associated response back to
| the client's browser allowing them to enjoy the creative
| and wonderful application we have prepared for them.
|
*/
$kernel = $app->make(Illuminate\Contracts\Http\Kernel::class);

$response = $kernel->handle(
    $request = Illuminate\Http\Request::capture()
);

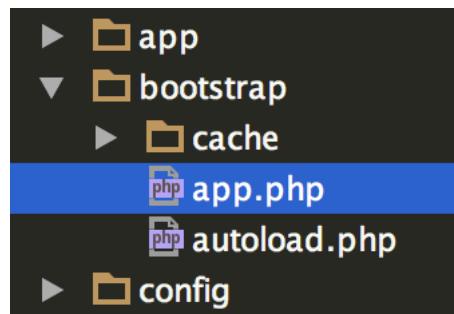
$response->send();

$kernel->terminate($request, $response);
```

The comments describe it exactly, so if you haven't done so already, please take a moment to read through them. You can see we pull in the autoloader and boot an instance of the app by requiring it:

```
$app = require_once __DIR__ . '/../bootstrap/app.php';
```

To get a sense of the \$app instance, we can go to our bootstrap folder and open app.php:



Here is the file, which has more comments than code, so again read the comments:

```
<?php

/*
|--------------------------------------------------------------------------
| Create The Application
|--------------------------------------------------------------------------
|
| The first thing we will do is create a new Laravel application instance
| which serves as the "glue" for all the components of Laravel, and is
| the IoC container for the system binding all of the various parts.
|
*/
$app = new Illuminate\Foundation\Application(
    realpath(__DIR__.'/../'))
;

/*
|--------------------------------------------------------------------------
| Bind Important Interfaces
|--------------------------------------------------------------------------
|
| Next, we need to bind some important interfaces into the container so
| we will be able to resolve them when needed. The kernels serve the
| incoming requests to this application from both the web and CLI.
|
*/
$app->singleton(
    Illuminate\Contracts\Http\Kernel::class,
```

```
App\Http\Kernel::class
);

$app->singleton(
    Illuminate\Contracts\Console\Kernel::class,
    App\Console\Kernel::class
);

$app->singleton(
    Illuminate\Contracts\Debug\ExceptionHandler::class,
    App\Exceptions\Handler::class
);

/*
-----
/ Return The Application
-----
/
/ This script returns the application instance. The instance is given to
/ the calling script so we can separate the building of the instances
/ from the actual running of the application and sending responses.
/
*/
return $app;
```

The comments tell us a lot. We open the application instance:

```
$app = new Illuminate\Foundation\Application(
    realpath(__DIR__ . '/../'));
```

The next part is best described by the comments:

```
/*
-----
/ Bind Important Interfaces
-----
/
/ Next, we need to bind some important interfaces into the container so
/ we will be able to resolve them when needed. The kernels serve the
/ incoming requests to this application from both the web and CLI.
/
*/
```

Then we return the application instance:

```
return $app;
```

So if we want to see what that object is composed of, we can simply var_dump and comment out the return statement like so:

```
var_dump($app)

// return $app;
```

You will get something like the following:

```

object(Illuminate\Foundation\Application)#3 (30) { ["basePath":protected]=> string(35) "/Users/billk/var/www/sample-project" ["hasBeenBoo
["bootingCallbacks":protected]=> array(0) { } ["bootedCallbacks":protected]=> array(0) { } ["terminatingCallbacks":protected]=> array(0) { }
object(Illuminate\Events\EventServiceProvider)#2 (2) { ["app":protected]=> *RECURSION* ["defer":protected]=> bool(false) } [1]=> object
*RECURSION* ["defer":protected]=> bool(false) } [2]=> object(Illuminate\Routing\RoutingServiceProvider)#7 (2) { ["app":protected]=> *R
["loadedProviders":protected]=> array(3) { ["Illuminate\Events\EventServiceProvider"]=> bool(true) ["Illuminate\Log\LogServiceProvider"]=
["deferredServices":protected]=> array(0) { } ["monologConfigurator":protected]=> NULL ["databasePath":protected]=> NULL ["storagePat
["environmentFile":protected]=> string(4) "env" ["namespace":protected]=> NULL ["resolved":protected]=> array(0) { } ["bindings":protect
{ ["this"]=> object(Illuminate\Events\EventServiceProvider)#2 (2) { ["app":protected]=> *RECURSION* ["defer":protected]=> bool(false) }
bool(true) } ["log"]=> array(2) { ["concrete"]=> object(Closure)#6 (1) { ["this"]=> object(Illuminate\Routing\RoutingService
bool(shared)=> bool(true) } ["router"]=> array(2) { ["concrete"]=> object(Closure)#8 (2) { ["this"]=> object(Illuminate\Routing\RoutingService
bool(false) } ["parameter"]=> array(1) { ["$app"]=> string(10) "" } } ["shared"]=> bool(true) } ["url"]=> array(2) { ["concrete"]=> object(Clo
(2) { ["app":protected]=> *RECURSION* ["defer":protected]=> bool(false) } ["parameter"]=> array(1) { ["$app"]=> string(10) "" } } ["share
(2) { ["this"]=> object(Illuminate\Routing\RoutingServiceProvider)#7 (2) { ["app":protected]=> *RECURSION* ["defer":protected]=> bool(f
bool(true) } ["Psr\Http\Message\ServerRequestInterface"]=> array(2) { ["concrete"]=> object(Closure)#11 (2) { ["this"]=> object(Illuminate\A
["defer":protected]=> bool(false) } ["parameter"]=> array(1) { ["$app"]=> string(10) "" } } ["shared"]=> bool(false) } ["Psr\Http\Message\Re
["this"]=> object(Illuminate\Routing\RoutingServiceProvider)#7 (2) { ["app":protected]=> *RECURSION* ["defer":protected]=> bool(false)
bool(false) } ["Illuminate\Contracts\Routing\ResponseFactory"]=> array(2) { ["concrete"]=> object(Closure)#13 (2) { ["this"]=> object(Illumi
*RECURSION* ["defer":protected]=> bool(false) } ["parameter"]=> array(1) { ["$app"]=> string(10) "" } } ["shared"]=> bool(true) } ["Illum
(3) { ["static"]=> array(2) { ["abstract"]=> string(32) "Illuminate\Contracts\Http\Kernel" ["concrete"]=> string(15) "App\Http\Kernel" } ["this
"" ["$parameters"]=> string(10) "" } } ["shared"]=> bool(true) } ["Illuminate\Contracts\Console\Kernel"]=> array(2) { ["concrete"]=> object(
"Illuminate\Contracts\Console\Kernel" ["concrete"]=> string(18) "App\Console\Kernel" } ["this"]=> *RECURSION* ["parameter"]=> array(
["shared"]=> bool(true) } ["Illuminate\Contracts\Debug\ExceptionHandler"]=> array(2) { ["concrete"]=> object(Closure)#16 (3) { ["static"]=>
"Illuminate\Contracts\Debug\ExceptionHandler" ["concrete"]=> string(22) "App\Exceptions\Handler" } ["this"]=> *RECURSION* ["paramet

```

Ok, it's a giant object dump. I grabbed part of it and formatted for readability:

```

["serviceProviders":protected]=> array(3) { [0]=>
object(Illuminate\Events\EventServiceProvider)#2
(2) { ["app":protected]=> *RECURSION* ["defer":protected]=>
bool(false) } [1]=> object(Illuminate\Log\LogServiceProvider)#5
(2) { ["app":protected]=> *RECURSION* ["defer":protected]=>
bool(false) } [2]=> object(Illuminate\Routing
\RoutingServiceProvider)#7
(2) { ["app":protected]=> *RECURSION*
 ["defer":protected]=> bool(false) } }
["loadedProviders":protected]=> array(3)
{ ["Illuminate\Events\EventServiceProvider"]=> bool(true)
["Illuminate\Log\LogServiceProvider"]=> bool(true)
["Illuminate\Routing\RoutingServiceProvider"]=> bool(true) }

```

The whole point of the exercise is that we literally find the service providers in the app instance. The service providers are the building blocks of the application.

Before we forget, let's restore your bootstrap/app.php file to what it was before by deleting the var_dump and uncommenting the return statement:

```
return $app;
```

Ok, so we have some idea of the flow of the application, how it boots up, even if we can't digest the giant \$app object in its entirety. You don't have to memorize any of this either, we are just taking a guided tour through it.

In public/index.php, you can see that once the \$app object is set, we handle the request via the kernel, just like the comments tell us:

```
/*
-----
/ Run The Application
-----
/
/ Once we have the application, we can handle the incoming request
/ through the kernel, and send the associated response back to
/ the client's browser allowing them to enjoy the creative
/ and wonderful application we have prepared for them.
/
*/
```

And here is the actual code that gets executed:

```
$kernel = $app->make(Illuminate\Contracts\Http\Kernel::class);

$response = $kernel->handle($request = Illuminate\Http\Request::capture());

$response->send();

$kernel->terminate($request, $response);
```

Without knowing every detail of these classes and methods, we can use the syntax of the code to describe it simply like this:

The app makes an instance of the Kernel, which uses its handle method to capture the request and create a response object. The send method of the response returns the response to the browser, then the kernel cleans up by terminating the request and the response.

The \$app->make() method is how the \$app resolves the Kernel class out of the container. You'll note that the make method here is calling an interface. IlluminateContractsHttpKernel.php looks like this:

```
<?php

namespace Illuminate\Contracts\Http;

interface Kernel
{
    /**
     * Bootstrap the application for HTTP requests.
     *
     * @return void
     */
    public function bootstrap();

    /**
     * Handle an incoming HTTP request.
     *
     * @param \Symfony\Component\HttpFoundation\Request $request
     * @return \Symfony\Component\HttpFoundation\Response
     */
    public function handle($request);

    /**
     * Perform any final actions for the request lifecycle.
     *
     * @param \Symfony\Component\HttpFoundation\Request $request
     * @param \Symfony\Component\HttpFoundation\Response $response
     * @return void
     */
    public function terminate($request, $response);

    /**
     * Get the Laravel application instance.
     *
     * @return \Illuminate\Contracts\Foundation\Application
     */
    public function getApplication();
}
```

This is an example of coding to an interface, not a concrete class, which is design pattern Laravel follows a lot. But it begs the question: How the heck does Laravel know which concrete class to call?

Well, the answer is in the bootstrap/app.php file:

```
$app->singleton(  
    Illuminate\Contracts\Http\Kernel::class,  
    App\Http\Kernel::class  
)
```

The \$app->singleton method is binding the interface, IlluminateContractsHttpKernel::class, to a concrete class AppHttpKernel::class. Whenever we call IlluminateContractsHttpKernel::class, it returns that instance of AppHttpKernel::class.

Because it is a singleton, it will return the same instance, when called more than once, rather than instantiating a new instance.

Doing it this way obviously creates another layer of complexity because you have to create the binding, but there is a huge upside. If you want to change the implementation of the concrete class throughout your codebase, you only have to make the one change in the binding.

Let's say you wanted to substitute the concrete Kernel class for SomeOther class:

```
$app->singleton(  
    Illuminate\Contracts\Http\Kernel::class,  
    App\Http\SomeOther::class  
)
```

Now if you call IlluminateContractsHttpKernel::class, you get AppHttpSomeOther::class.

You only need to make the change here, once, as opposed to having to go through every instance of the class being called in your application. So this approach scales well.

Automatic Injection

With Laravel, we get automatic injection, in both methods and constructors. Automatic injection allows you to instantiate an instance of a class by type hinting it, instead of newing it up. Let's look at some examples.

Method Injection

Let's setup a super simple class. Inside of the app/Utilities folder, create a file named SomethingNew.php with the following contents:

Gist:

[SomethingNew.php](#)

From book:

```
<?php

namespace App\Utilities;

class SomethingNew
{
    public function display()
    {
        return 'something special';
    }
}
```

Next, in our app/Http/Controllers/TestController.php file, let's change the index method to the following:

Gist:

[index method](#)

From book:

```
public function index()
{
    $something = new SomethingNew;

    return $something->display();

}
```

Don't forget to add a use statement for the class:

```
use App\Utilities\SomethingNew;
```

Now when you visit sample-project.com/test, you will see:

```
something special
```

So that's just an example of newing up a class and using the instance variable. But with Laravel, you can do something like this:

Gist:

[index method](#)

From book:

```
public function index(Thing $something)
{
    return $something->display();
}
```

Laravel, using PHP reflection, resolves the class automatically for you. That's just way more readable and cleaner. It's another reason to love Laravel.

Constructor Injection

If you create a constructor for TestController and try the same code in the constructor, it works there too.

```
public function __construct(Thing $something)
{
    return $something;
}
```

Laravel is doing all this for you without you having to create a binding in a service provider.

Service Providers

When you want to code to a contract instead of a concrete class or your class has dependencies, then you'll want to use a service provider for the binding.

Out of the box, Laravel comes with an app service provider, which we already used for our view composer.

A service provider is not something I normally reach for in my personal development, so to demonstrate, I'm just going to create a dummy implementation.

Let's start by adding a Contracts folder inside of our Utilities folder like so:



Inside of the folder, let's create RocketShipContract.php with the following contents:

Gist:

[RocketShipContract.php](#)

From book:

```
<?php

namespace App\Utilities\Contracts;

interface RocketShipContract
{

    public function blastOff();

}
```

So all the contract is doing is forcing whichever class implements it to have a public blastOff method.

Next, in the Utilities folder, let's create the following two files:

Gist:

[RocketShip.php](#)

From book:

```
<?php

namespace App\Utilities;

use App\Utilities\Contracts\RocketShipContract;

class RocketShip implements RocketShipContract
{

    public function blastOff()
    {

        return 'Houston, we have ignition';

    }

}
```

In the same folder:

Gist:

[SpaceShip.php](#)

From book:

```
<?php

namespace App\Utilities;

use App\Utilities\Contracts\RocketShipContract;

class SpaceShip implements RocketShipContract
{

    public function blastOff()
    {

        return 'to the moon';

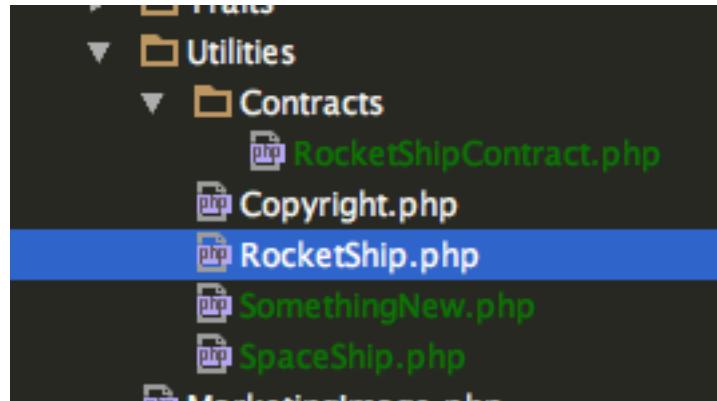
    }

}
```

```
}
```

```
}
```

So now your Utilities folder looks like this:



Ok, next we are going to create the RocketShipServiceProvider by running the following from your command line:

```
php artisan make:provider RocketShipServiceProvider
```

Laravel will of course put the new file in the app/Providers folder for you:

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;

class RocketShipServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap the application services.
     *
     * @return void
     */
}
```

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

/**
 * Register the application services.
 *
 * @return void
 */
public function register()
{
    //
}

}
```

You can see it's just got couple of stubs on it. Let's change the register method to the following:

Gist:

[RocketShipServiceProvider.php](#)

From book:

```
public function register()
{
    $this->app->bind(
        'App\Utilities\Contracts\RocketShipContract',
        'App\Utilities\RocketShip'
    );
}
```

The gist has the full file for reference.

You can see that we are using the bind method, and giving it two parameters. The first is the contract and the second is the concrete implementation.

Alternatively, we could have written the method this way:

```
$this->app->bind('App\Utilities\Contracts\RocketShipContract', function() {  
    return new RocketShip();  
});
```

Sorry I had to break the line in the bind method in order to avoid the wordwrap. That would obviously be one line.

Also note that you can pass an instance of \$app into the closure, as in this example from the [docs](#):

```
$this->app->singleton('HelpSpot\API', function ($app) {  
    return new HelpSpot\API($app->make('HttpClient'));  
});
```

Ok, one last step before we use our new binding. We need to register the provider in our config/app.php file providers array:

```
/*
 * Application Service Providers...
 */
App\Providers\AppServiceProvider::class,
App\Providers\AuthServiceProvider::class,
App\Providers\BroadcastServiceProvider::class,
App\Providers\EventServiceProvider::class,
App\Providers\RouteServiceProvider::class,
App\Providers\RocketShipServiceProvider::class,
```

You can see it's the last line that we added.

Ok, now let's go to our TestController. Let's add the use statement:

```
use App\Utilities\Contracts\RocketShipContract;
```

Let's empty the constructor so it looks like this:

```
public function __construct()
{
}
```

We'll keep the stub around in case we need to test something with a constructor in the future.

And finally, let's change the index method to the following:

Gist:

[index method](#)

From book:

```
public function index(RocketShipContract $rocket)
{
    return $rocket->blastOff();
}
```

Now hit the test route:

`sample-project.com/test`

And you should see:

`Houston, we have ignition`

So we have our concrete RocketShip class bound to our contract, so that when we type hint our contract, we get the class we want.

One of the main points of doing this is the ease at which you can switch implementations. Let's go to our RocketShipServiceProvider and change the register method to the following:

```
public function register()
{
    $this->app->bind(
        'App\Utilities\Contracts\RocketShipContract',
        'App\Utilities\SpaceShip'
    );
}
```

We don't need a gist because we made a one word change, swapping in Space for Rocket, so that we bind the contract to SpaceShip.php.

Now when we hit our test route again:

sample-project.com/test

You should see:

```
regretfully, we do not have ignition
```

And that's it!

Our service provider implementation demonstrated coding to an interface and binding it to a concrete class, which makes swapping out implementations easy. This is what is meant by loose coupling and it makes the code less brittle.

We have one more property we can add to our service provider. Let's drop in the following:

```
protected $defer = true;
```

This insures that the provider is only loaded when it is needed.

You'll also note that we left the boot method empty. The boot method provides instructions after the app has loaded, like in the example we did with the view composer in the AppServiceProvider:

```
public function boot()
{
    view()->composer('layouts.bottom', function($view){
        $view->with('copyright',
            \App\Utilities\Copyright::displayNotice());
    });
}
```

Note, you can't set the defer property to true if there is something in the boot method.

One reason why you might create a service provider is to handle dependencies that would be awkward to type over and over in your code, and again much simpler to change if you have them aliased in one place.

So to demonstrate, let's change our RocketShip class to the following:

Gist:

[RocketShip.php](#)

From book:

```
<?php

namespace App\Utilities;

use App\Utilities\Contracts\RocketShipContract;

class RocketShip implements RocketShipContract
{

    public $fuelTank;

    public $oxygen;

    public function __construct(FuelTank $fuelTank, Oxygen $oxygen)
    {

        $this->fuelTank = $fuelTank;

        $this->oxygen = $oxygen;

    }

    public function blastOff()
    {

        return 'Houston, we have ignition';

    }

}
```

All we did is add two dependencies, FuelTank and Oxygen, and set them as class properties. Those classes don't exist, so we'll create them now in our app/Utilities folder:

Gist:

[FuelTank.php](#)

From book:

```
<?php

namespace App\Utilities;

class FuelTank
{



}
```

It's just an empty stub, we are not concerned with the content of the class or what it does, just the stitching of how all this fits together.

Gist:

[Oxygen](#)

From book:

```
<?php

namespace App\Utilities;

class Oxygen
{



}
```

Again it's just an empty stub.

Ok, so now we are ready to play with this in our RocketShipServiceProvider. Let's change it to the following:

Gist:

[RocketShipServiceProvider](#)

From book:

```
<?php

namespace App\Providers;

use App\Utilities\RocketShip;
use Illuminate\Support\ServiceProvider;
use App\Utilities\Oxygen;
use App\Utilities\FuelTank;

class RocketShipServiceProvider extends ServiceProvider
{

    protected $defer = true;

    /**
     * Bootstrap the application services.
     *
     * @return void
     */

    public function boot()
    {
        //
    }

    /**
     * Register the application services.
     *
     * @return void
     */

    public function register()
    {
        $this->app->bind(
            'App\Utilities\Contracts\RocketShipContract',
            function(){

                return new RocketShip(new FuelTank(), new Oxygen());
            });
    }
}
```

```
}
```

This time, the second argument of the bind method is a closure, and this is where we instantiate the class with its dependencies.

And when you hit your test url, you should see:

```
Houston, we have ignition
```

So it's all working fine, but we can do one more refinement. RocketShipContract might be a little tedious to type. In that case, we can set up an alias.

Aliases

In your RocketShipServiceProvider.php file, add the following property to the class:

```
protected $aliases = [  
    'Rocket' => 'App\Utilities\Contracts\RocketShipContract'  
];
```

Then in config/app.php, in your aliases array, add the following line:

```
'Rocket' => App\Utilities\Contracts\RocketShipContract::class,
```

Ok, let's go back to our TestController. Remove the use statement for the contract:

```
use App\Utilities\Contracts\RocketShipContract;
```

Now add the following use statement:

```
use Rocket;
```

Now change the index method to the following:

```
public function index(Rocket $rocket)
{
    return $rocket->blastOff();
}
```

No gist, we're only changing the type hint to Rocket.

And when you hit your test url again, you should see:

Houston, we have ignition

How cool is that? Hopefully you are gaining an understanding of how Laravel uses aliases to make the code easier to work with and maintain. You can read through your aliases array in config/app to see all the aliases that are being used in your app.

Facades In Realtime

Ok, let's look at realtime facades, which is new with Laravel 5.4. Let's go into config/app.php and comment out in your providers array:

```
//App\Providers\RocketShipServiceProvider::class,
```

And in the same file, in your aliases array, let's comment out:

```
//'Rocket' => App\Utilities\Contracts\RocketShipContract::class,
```

That will disable our RocketShipServiceProvider, so we can play with the facade. Let's change TestController.php to the following:

Gist:

[TestController.php](#)

From book:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Facades\App\Utilities\RocketShip;

class TestController extends Controller
{

    public function index()
    {
        return RocketShip::blastOff();
    }
}
```

So we actually have less code in the controller. Let's start with the use statement:

```
use Facades\App\Utilities\RocketShip;
```

By adding Facades in front, Laravel will build a facade for us on the fly, pulling in the dependencies.

That means we can call our RocketShip class directly without typehinting:

```
return RocketShip::blastOff();
```

Now hit your test url again, you should see:

```
Houston, we have ignition
```

You'll note that blastOff is not a static method, and yet we can call it as if it were. This opens up a lot of possibilities when you are coding. It's just awesome. It's so new, I haven't had time to digest it in my coding, but I'm really looking forward to being able to do so.

I should also note that the dynamic facade is not quite the same as coding to a contract. In this case, we are calling the concrete class RocketShip, which implements the contract.

When we were using the service provider, we bound RocketShip to RocketShipContract, so whenever we called RocketShipContract or the Rocket alias we created for it, we were calling RocketShip or SpaceShip, depending on which class we defined in the bind method in the service provider.

If you want to read more about Laravel's service providers, check out the [service provider docs](#). Also, there is a good thread in the Laracasts forum with the subject of [Why Use Service Providers](#), which talks about the advantages of using a service provider and when it would be appropriate.

Summary

In this chapter, we had a lot of fun with sending mail, creating events, and learning about the service container, service providers, aliases, and dynamic facades.

In the next chapter, we are going to go a little deeper into intermediate programming by implementing a basic chatroom with Vue.js, Laravel Echo, and Pusher. It will be a lot of fun. See you there.

Chapter 14 Chat with Laravel Echo, Vue, and Pusher

This is an exciting chapter for us because we will be touching on a number of subjects, leveling up our programming skills. We will of course build a basic chat application, which will utilize Laravel Echo, Vue.js and Pusher.

Providing real-time feedback to the user is expected these days and developing a chat app is a great example of this. We will also get a chance to use nested components in Vue.js, learning how events traverse the DOM in them.

We will also be using some ES6, also known as Echmascript 2015, the latest version of javascript, which is not yet supported by all browsers. We don't have to worry about that though, because we are using Laravel Mix, which will compile everything down for us.

Vue.js Nested Components

This is a fairly deep subject, so here is a link to the [Vue Component Docs](#) in case you want to reference them.

We're going to start with a super simple example. We going to create a component that lists items. It's not going to do much, but we can use it to see how the parent/child components stitch together.

Let's start by changing our resources/assets/js/app.js file to the following:

Gist:

[app.js](#)

From book:

```
/**  
 * First we will load all of this project's JavaScript dependencies which  
 * includes Vue and other libraries. It is a great starting point when  
 * building robust, powerful web applications using Vue and Laravel.  
 */  
  
require('./bootstrap');  
  
/**  
 * Next, we will create a fresh Vue application instance and attach it to  
 * the page. Then, you may begin adding components to this application  
 */
```

```
* or customize the JavaScript scaffolding to fit your unique needs.  
*/  
  
Vue.component('example', require('./components/Example.vue'));  
Vue.component('widget-grid', require('./components/WidgetGrid.vue'));  
Vue.component('marketing-image-grid',  
require('./components/MarketingImageGrid.vue'));  
Vue.component('parent', require('./components/Parent.vue'));  
Vue.component('child', require('./components/Child.vue'));  
Vue.component('add-item', require('./components/AddItem.vue'));  
  
const app = new Vue({  
  
  el: '#app',  
  
  data: {  
  
    items: ['apples', 'berries', 'bananas']  
  },  
  
  methods: {  
  
    addItem(item){  
  
      // add to existing messages  
      this.items.push(item.item);  
    }  
  }  
});
```

You can see that we added component calls that we do not have yet:

```
Vue.component('parent', require('./components/Parent.vue'));
Vue.component('child', require('./components/Child.vue'));
Vue.component('add-item', require('./components/AddItem.vue'));
```

So don't try to run Mix, it will return an error.

Let's look at what we added to the root instance of Vue:

```
const app = new Vue({
  el: '#app',
  data: {
    items: ['apples', 'berries', 'bananas']
  },
  methods: {
    addItem(item){
      // add to existing messages
      this.items.push(item.item);
    }
  }
});
```

You can see we have an `items` property in `data` as well as a method to push items into the array. So now we can imagine what our components will be.

We'll have a parent that lists the items. We'll have a child that formats each item. And we'll have a separate component that adds new items.

So the next step is to place the components in a view. We'll use our handy test/index.blade.php view. Let's change that to the following:

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

@section('title')
    <title>Test Page</title>

@endsection

@section('content')
    <h1>Fruit Basket</h1>
    <div>
        <parent v-bind:items="items"></parent>
        <add-item v-on:itemcreated="addItem"></add-item>
    </div>
@endsection
```

You can see we added the components. Again, this will not work because we have not created the components yet. But it's worth talking about what we are doing.

You can see on the parent we are binding items to the component via v-bind:

```
<parent v-bind:items="items"></parent>
```

Alternatively, you can just use the colon, for example, `:items`, but I prefer to use `v-bind` because I'm half blind and I might miss the colon by itself when I'm reviewing code whereas `v-bind` is hard to miss.

So the first thing to understand is that properties flow from the root Vue instance down to the component. The items we are binding to the component come from the data we have defined on the root instance:

```
data: {  
    items: ['apples', 'berries', 'bananas']  
},
```

And on the Parent component, we will have to declare the properties via props, like so:

```
<script>  
    export default {  
        props: ['items'],  
    }  
  
</script>
```

This makes the properties that were bound on the component tag available to the script. The other component tag we have on test/index.blade.php view is:

```
<add-item v-on:itemcreated="addItem"></add-item>
```

This component tag calls the add-item form, which we will use to add items to our list. The thing to note here is that we have a v-on directive to listen for an event named “itemcreated”, which will fire the addItem method on the root instance.

Where properties flow from the root down, events flow from the component up to the root instance via \$emit and the v-on listener, we will see in a moment when we create the components.

Before we do that, let’s make sure our index method on our TestController is as follows:

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

All we want is to call the index page, our components will do the rest.

Ok, let’s create the component files in resources/assets/js/components:

Gist:

[Parent.vue](#)

From book:

```
<template lang="html">

  <div class="child">
    <ul>
      <child v-for="item in items" v-bind:item="item"></child>
    </ul>
  </div>

</template>

<script>
  export default {
    props: [ 'items' ],
  }
</script>
```

Ok, so Parent receives the props that we bound to it on the component tag:

```
<script>
  export default {
    props: [ 'items' ],
  }
</script>
```

In the template part, we have a nested component. In this case, it calls the child component:

```
<child v-for="item in items" v-bind:item="item"></child>
```

We use the v-for directive to iterate through the items, which we get from props, then we use the local variable "item" to bind to the child component. We will see how that is used next.

Gist:

[Child.vue](#)

From book:

```
<template lang="html">

  <div class="item">

    <div id="item">

      <li>{{ item }}</li>

    </div>

  </div>

</template>

<script>

  export default {

    props: [ 'item' ],

  }

</script>
```

Because we bound the item as a property, we can use it. We don't want items because we want to show each individual item. Then we simply put item between tags and we're set:

```
<li>{{ item }}</li>
```

To add an item to our list, we have a separate component.

Gist:

[AddItem.vue](#)

From book:

```
<template lang="html">

  <div class="add-item">

    <input type="text"
      placeholder="add item..."
      v-model="item"
      @keyup.enter="addSingleItem">

    <button class="btn btn-primary"
      @click.prevent="addSingleItem">
      Send
    </button>

  </div>

</template>

<script>
```

```
export default {

  data() {
    return {
      item: ''
    }
  },
  methods: {
    addSingleItem(){
      this.$emit('itemcreated', {
        item: this.item,
      });
      this.item = '';
    }
  }
}

</script>

<style lang="css">
.add-item{
  margin: 2rem;
  display: flex;
}
.add-item input {
```

```
border: 1px solid;
flex: 1 auto;
padding: 1rem;

}

.add-item button {

border-radius: 0;

}

</style>
```

There's a little more to this one. Let's start with the script:

```
<script>

export default {

  data() {
    return {
      item: ''
    }
  },
  methods: {
    addSingleItem(){
      this.$emit('itemcreated', {
        item: this.item,
      })
    }
  }
}

</script>
```

```
});  
  
    this.item = '';  
}  
  
}  
  
</script>
```

This time, we are not bringing in any properties, since we are creating the item. We give it an initial value using the data method:

```
data() {  
  
    return {  
  
        item: ''  
    }  
},
```

Note that this is ES6, plain javascript would like the following:

```
Data: function() {  
  
    return {  
  
        item: ''  
  
    }  
  
},
```

Personally I like the new syntax and will start using it more.

So the other part of the script is the addSingleItem method. I called it that because on the root instance I have addItem and I didn't want to get confused about which method I was working with. Here is the method:

```
addSingleItem(){  
  
    this.$emit('itemcreated', {  
  
        item: this.item,  
  
    });  
  
    this.item = '';  
  
}
```

We are using the \$emit method to broadcast the event up the DOM to the root instance. The name of the event is set to 'itemcreated' and is the first parameter of the method. The second param is an object. We have item set to this.item, which has been set from the input on the form. Then we reset the form by setting the value of this.item to an empty string.

Now let's look at the input form in the template:

```
<div class="add-item">

  <input type="text"
    placeholder="add item..."
    v-model="item"
    @keyup.enter="addSingleItem">

  <button class="btn btn-primary"
    @click.prevent="addSingleItem">
    Send
  </button>

</div>
```

We are binding the value of item to the input via v-model. On keyup.enter, we call the method addSingleItem, which fires the “itemcreated” event. We can also call the method by clicking the send button, which fires the event as well.

I added some styling in the style section:

```
<style lang="css">

.add-item{

  margin: 2rem;
  display: flex;

}

.add-item input {

  border: 1px solid;
  flex: 1 auto;
  padding: 1rem;

}
```

```
.add-item button {  
  
    border-radius: 0;  
}  
  
</style>
```

It's cool that you get to work on everything in one place. But I have heard from some enterprise programmers that on large applications, this is not really practical. You will have to decide for yourself which way you want to go. For the purposes of our work here, we will keep it all in one file.

Ok, you should be able to compile this down by running:

```
npm run dev
```

Alternatively, you could run:

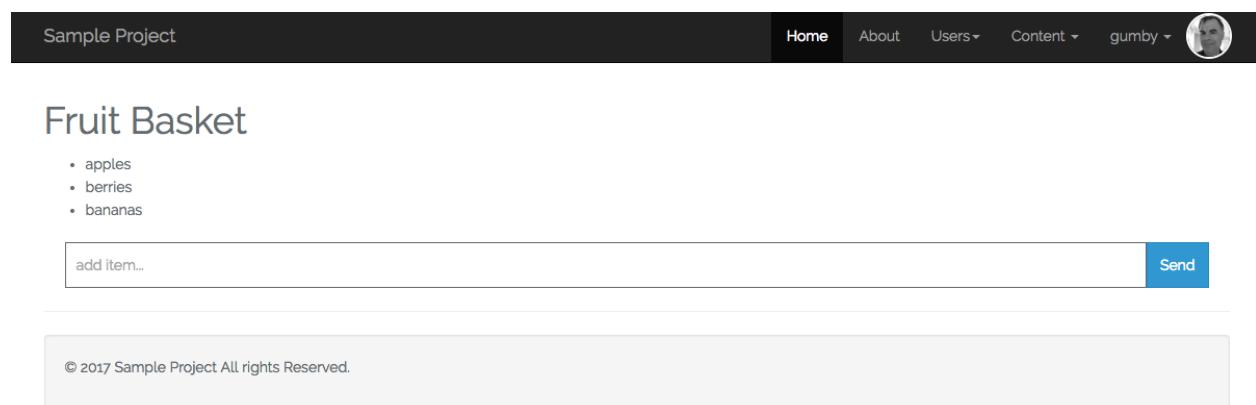
```
npm run watch
```

That will listen for changes in your assets files and automatically run for you. It's pretty handy. Use control C to exit that when you want to turn it off.

Now if you got to your test route:

```
sample-project.com/test
```

You should see something like the following:



The screenshot shows a web application interface. At the top, there is a dark header bar with the text "Sample Project" on the left and navigation links "Home", "About", "Users", "Content", and "gumby" on the right, along with a user profile icon. Below the header, the main content area has a title "Fruit Basket". Underneath the title is a list of items: "apples", "berries", and "bananas", each preceded by a small circular bullet point. Below this list is a horizontal input field containing the placeholder text "add item...". To the right of the input field is a blue rectangular button with the word "Send" in white. At the bottom of the page, there is a light gray footer bar with the copyright notice "© 2017 Sample Project All rights Reserved."

Chat

Ok, we're ready to start working on our chat. The good news is that a lot of the principles we learned from our fruit basket example will apply.

Also, I should note that I followed the [video tutorial series](#) by Josh Larsen. He does an excellent job of building up the application from scratch. If you watch it, it will reinforce what you are learning here.

In my example, the basic approach is the same, but we are doing some things differently, refinements like making the retrieval of the username more robust and handing in the date time, so that users can see the time and date of their messages.

Throughout this example, we will reinforce the idea that properties flow down from the root instance, while events flow up to the root instance from the component.

Routes

Let's start by adding some routes to routes/web.php that we are going to need:

Gist:

[web.php](#)

From book:

```
// Chat routes

Route::get('/chat-messages', 'ChatController@getMessages')
    ->middleware('auth');

Route::post('/chat-messages', 'ChatController@postMessage')
    ->middleware('auth');

Route::get('/chat', 'ChatController@index')->middleware('auth');

// Username route

Route::get('/username', 'UsernameController@show')
    ->middleware('auth');
```

You can add these routes to your routes file with the comments in alphabetical order. The gist will be the entire web.php file.

Just a heads up, when you try to run your application with routes to controllers and actions that don't exist, it will return an error. Since we still need to create these controllers referenced in the new routes, your application will not run.

Chat Controller

Let's get everything in place. We start with the ChatController:

Gist:

[ChatController.php](#)

From book:

```
<?php

namespace App\Http\Controllers;

use App\Exceptions\UnauthorizedException;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;
use App\Message;

class ChatController extends Controller
{

    public function index()
    {

        return view('chat.index');

    }

    public function getMessages(Request $request)
    {

        if ( ! $request->ajax()) {

            throw new UnauthorizedException();

        }

    }

}
```

```
}

$messages = Message::with('user')->MostRecent()->get();

$messages = array_reverse($messages->toArray());

return $messages;

}

public function postMessage(Request $request)
{
    if ( ! $request->ajax()) {

        throw new UnauthorizedException();
    }

$user = Auth::user();

$message = $user->messages()->create([
    'message' => request()->get('message')
]);

//broadcast(new MessagePosted($message, $user))->toOthers();

return ['status' => 'OK'];
}

}
```

Before we can go over the controller, it depends on the Message model, which we have not created yet. So let's run from the command line:

```
php artisan make:model Message -m
```

Message Migration

Let's do the migration first. Let's change the migration file to the following:

Gist:

[messages migration](#)

From book:

```
<?php

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

class CreateMessagesTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */

    public function up()
    {
        Schema::create('messages', function (Blueprint $table) {

            $table->increments('id');
            $table->text('message');
            $table->integer('user_id')->unsigned();
            $table->timestamps();

        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
}
```

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

}


```

All we are really doing is adding the message and the user_id, which we will use to identify the message author.

Message Model

The next step is to change the model, Message.php, file to the following.

Gist:

[Message.php](#)

From book:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Message extends Model
{
    protected $fillable = ['message', 'user_id'];

    public function scopeMostRecent($query)
    {

        return $query->orderBy('created_at', 'desc')->limit(10);

    }
}
```

```
public function user()
{
    return $this->belongsTo(User::class);
}

}
```

You can see we made message and user_id fillable.

The next thing you see is a query scope:

```
public function scopeMostRecent($query)
{
    return $query->orderBy('created_at', 'desc')->limit(10);
}
```

This allows us to write the following to retrieve what we want:

```
Message::with('user')->mostRecent()->get();
```

There's a lot of power in that one line. First, we are eager loading the related User records that the messages belong to.

Eager Loading

Eager loading alleviates what is known as the n+1 problem, which is when a separate query runs for each pass through a loop. For example:

```
$messages = Message::MostRecent()->get();  
  
foreach ($messages as $message) {  
  
    echo $message->user->name;  
  
}
```

In this example, we are using the relationship to pull the user's name inside the foreach loop, but we have to do a separate query for each pass of the loop.

When we write it like this:

```
$messages = Message::with('user')->MostRecent()->get();  
  
foreach ($messages as $message) {  
  
    echo $message->user->name;  
  
}
```

In this case the associated user records are already part of the object, so no additional queries are needed.

Query Scopes

At this point, you probably noticed how beautiful the scope syntax is. When you create a scope, you define it on the model. You use scope as the first word, then whatever you want to follow.

Here's a couple examples from the [docs](#):

```
/*
 * Scope a query to only include popular users.
 *
 * @param \Illuminate\Database\Eloquent\Builder $query
 * @return \Illuminate\Database\Eloquent\Builder
 */
public function scopePopular($query)
{
    return $query->where('votes', '>', 100);
}

/*
 * Scope a query to only include active users.
 *
 * @param \Illuminate\Database\Eloquent\Builder $query
 * @return \Illuminate\Database\Eloquent\Builder
 */
public function scopeActive($query)
{
    return $query->where('active', 1);
}
```

Let's finish up with the Message model. We have a relationship to the user:

```
public function user()
{
    return $this->belongsTo(User::class);
}
```

Messages Method On User Model

While we're thinking of it, let's add the messages relationship to User.php.

Gist:

[messages method on User.php](#)

From book:

```
public function messages()
{
    return $this->hasMany(Message::class);
}
```

Ok, that's pretty straightforward.

Messages Factory

Next let's create a factory method in database/factories/ModelFactory.php:

Gist:

[Message Factory](#)

From book:

```
$factory->define(App\Message::class, function ($faker) {
    $message = $faker->unique()->word . ' '
        . $faker->unique()->word;

    return [
        'message' => $message,
        'user_id' => 1
    ];
});
```

Then let's go ahead and run from the command line:

```
php artisan tinker
```

Then run:

```
factory('App\Message', 30)->create();
```

Ok, we're almost ready for a test. Let's go to ChatController and comment out the if statement, so that the getMessages method looks like this:

```
public function getMessages(Request $request)
{
    /**
     *
     * if ( ! $request->ajax() ) {
     *
     *     throw new UnauthorizedException();
     *
     * }
     *
     */
    $messages = Message::with('user')->MostRecent()->get();
    $messages = array_reverse($messages->toArray());
    return $messages;
}
```

Since we commented the test to make sure the request was an ajax call, we check via the browser to see if we are getting results.

Type in the following url into your browser:

```
sample-project.com/chat-messages
```

You should get json results:

```
[  
  {  
    "id": 164,  
    "message": "sequi eaque",  
    "user_id": 1,  
    "created_at": "2017-02-06 01:37:23",  
    "updated_at": "2017-02-06 01:37:23",  
    "user": {  
      "id": 1,  
      "name": "gumby",  
      "email": "smapple@sample.com",  
      "is_subscribed": 0,  
      "is_admin": 1,  
      "status_id": 10,  
      "created_at": "2017-01-24 23:31:42",  
      "updated_at": "2017-01-28 02:18:48"  
    }  
  },  
  {  
    "id": 163,  
    "message": "alias sunt",  
    "user_id": 1,  
    "created_at": "2017-02-06 01:37:23",  
    "updated_at": "2017-02-06 01:37:23",  
    "user": {  
      "id": 1,  
      "name": "gumby",  
      "email": "smapple@sample.com",  
      "is_subscribed": 0,  
      "is_admin": 1,  
      "status_id": 10,  
      "created_at": "2017-01-24 23:31:42",  
      "updated_at": "2017-01-28 02:18:48"  
    }  
  },  
  {  
    "id": 162,  
    "message": "quos voluptates",  
  }
```

Note, your formatting may differ if you do not have the json formatter chrome plugin.

Anyway, you can see we are able to pull results. We have our messages eager loaded with the associated user.

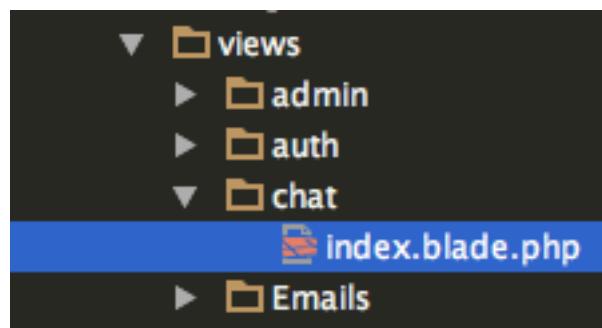
Let's go back and remove the comment from the ChatController getMessages method. Once you done that, hit your url again to make sure we are getting the expected behavior. You should see an Unauthorized exception because we only want that request to come in via ajax.

Chat Front End

We're making excellent progress and we're ready to start working on the front end.

Chat Index

We need to create a chat folder in our views directory, and within that, an index.blade.php file:



Let's put the following contents into your chat/index.blade.php file.

Gist:

[index.blade.php](#)

From book:

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

@section('content')

<div class="panel panel-default">

    <div class="panel-heading">

        Chatroom

    </div>
```

```
</div>

<chat-list v-bind:messages="messages"></chat-list>

<chat-create v-on:messagecreated="addMessage"
             :currentuser="currentuser"></chat-create>

</div>

@endsection
```

Ok, so logically, we are going to have some kind of chat list, where we bind the messages, much like we did the items to the parent component we made in the previous example:

```
<chat-list v-bind:messages="messages"></chat-list>
```

So we know we will be operating on the messages.

Next we have the chat-create component:

```
<chat-create v-on:messagecreated="addMessage"
             :currentuser="currentuser"></chat-create>

</div>
```

You can see it's listening for a messagecreated event, and when it receives one, it will call the addMessage method, which will be on the root instance.

We are also binding in the currentuser as a property. If any of this is fuzzy, don't worry, it will become clear as we create our root Vue instance and components.

components.js

Since our list of components is getting longer, I'm going to create a separate file to hold them. Create a new js file at resources/assets/js/components.js with the following contents:

Gist: [components.js](#)

From book:

```
Vue.component('example',
require('./components/Example.vue'));
Vue.component('widget-grid',
require('./components/WidgetGrid.vue'));
Vue.component('marketing-image-grid',
require('./components/MarketingImageGrid.vue'));
Vue.component('chat-message',
require('./components/ChatMessage.vue'));
Vue.component('chat-list',
require('./components/ChatList.vue'));
Vue.component('chat-create',
require('./components/ChatCreate.vue'));
```

Obviously use the gist for formatting. Also note that I did not include the components from our fruit item list example. They require a different root instance, and we're not planning on using that code, so there is no need to call in the components.

That said, you can see how easy it will be to add new components to our list. This helps keep our app.js file from getting overstuffed. We should seriously consider extracting out the root view instance into its own file as well, but to keep it simpler, we will leave it as is for now.

app.js

The next step is to change resources/assets/js/app.js to the following:

Gist:

[app.js](#)

From book:

```
/**  
 * First we will load all of this project's JavaScript dependencies which  
 * includes Vue and other libraries. It is a great starting point when  
 * building robust, powerful web applications using Vue and Laravel.  
 */  
  
require('./bootstrap');  
  
/**  
 * We will require in our components.js file, which contains our  
 * component files. Putting them in their own file reduces clutter.  
 */  
  
require('./components');  
  
/**  
 * Next, we will create a fresh Vue application instance and attach it to  
 * the page. Then, you may begin adding components to this application  
 * or customize the JavaScript scaffolding to fit your unique needs.  
 */  
  
const app = new Vue({  
    el: '#app',  
  
    data: {  
  
        messages: [],  
  
        currentuser: '',  
  
        roomCount: []  
  
    },  
  
    methods: {  
  
        addMessage(message) {  
  
            // add to existing messages  
  
            this.messages.push(message);  
  
            axios.post('/chat-messages', message)  
    }  
});
```

```
.then(response => {  
}  
}  
.catch(error => {  
  console.log(error);  
});  
  
}  
  
},  
  
created(){  
  axios.get('/chat-messages').then(response=> {  
    this.messages = response.data;  
  });  
  
  axios.get('/username').then(response=> {  
    this.currentuser = response.data;  
  });  
}  
};  
});
```

Obviously, we are requiring bootstrap like before, and now we are pulling in our components file as well. Then it's onto the root vue instance.

So let's take a look at:

```
const app = new Vue({  
  
  el: '#app',  
  
  data: {  
  
    messages: [],  
  
    currentuser: '',  
  
    roomCount: []  
  
  },  
  _____
```

You can see we have an array for our messages, a currentuser with a default value of an empty string, and a roomCount array.

The roomCount will hold the number of current users connected to the chat room, but we are going to ignore that for now, since that will be part of our Laravel Echo implementation.

Next up we have our methods:

```
_____
```

```
methods: {  
  
  addMessage(message) {  
  
    // add to existing messages  
  
    this.messages.push(message);  
  
    axios.post('/chat-messages', message)  
      .then(response => {  
  
        })  
      .catch(error => {  
  
        console.log(error);  
      });  
  };  
  _____
```

```
}
```

```
,
```

This consists of a single method, addMessage, that takes in message as an argument. We push the new message onto the messages data array, then we use the axios library to make a post to the server.

In the grids we built, we used jquery, but the axios library is better suited to what we want. You can see it uses promises because we use the then method to handle the callback.

This is part of ES6, and it's a great feature. I recommend reading up on ES6 on your own. We're not doing anything with the response after hitting the '/chat-messages' post route.

If we go to the corresponding controller action on ChatController, which is the postMessage method, we see the following:

```
public function postMessage(Request $request)
{
    if ( ! $request->ajax() ) {

        throw new UnauthorizedException();

    }

    $user = Auth::user();

    $message = $user->messages()->create([
        'message' => request()->get('message')
    ]);

    //broadcast(new MessagePosted($message, $user))->toOthers();

    return ['status' => 'OK'];
}
```

You can see we check to see if it's an ajax request, if not, throw an Unauthorized exception. Then we get the authenticated user:

```
$user = Auth::user();
```

Next we get the nested object, using the messages relationship on the User model to create the message record:

```
$message = $user->messages()->create([
    'message' => request()->get('message')
]);
```

We'll skip over the commented line, we will be using that in the future, when we use Laravel Echo. For now, we just return:

```
return ['status' => 'OK'];
```

Ok, so returning to our root vue instance, we move on to the created method:

```
created(){
    axios.get('/chat-messages').then(response=> {
        this.messages = response.data;
    });

    axios.get('/username').then(response=> {
        this.currentuser = response.data;
    });
}
```

I was always a little confused about the lifecycle hooks and the differences between created and mounted. I found an awesome diagram in the [Vue.js docs](#), you should check that out when you get a chance. It will give you a much better understanding of the lifecycle hooks and show you where created occurs as opposed to mounted.

Our created function is doing 2 things. First it grabs the messages in our db via an axios get request. Second, it returns the current user's username with a second axios call to '/username.'

The get route points to a UsernameController, but we have not made that yet. Let's do so now with the following on the command line:

```
php artisan make:controller UsernameController
```

Then let's change the contents of UsernameController.php to the following.

Gist:

[UsernameController.php](#)

From book:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;
use App\Exceptions\UnauthorizedException;

class UsernameController extends Controller
{
    public function show()
    {
        if (! request()->ajax()){

            throw new UnauthorizedException();

        }

        return Auth::user()->name;
    }
}
```

Not much to it, we just return the username of the current user.

So our axios ajax request will return this value and we set it on our data object as this.currentuser:

```
axios.get('/username').then(response=> {

    this.currentuser = response.data;

});
```

I should mention that we are using ES6 again:

```
.then(response => {  
  
  this.currentuser = response.data;  
  
})
```

This is equivalent to:

```
.then(function(response){  
  
  this.currentuser = response.data;  
  
})
```

I'm starting to really like the new syntax, and sometimes I forget it's new, which is a good sign.

Ok, so we have a good idea of what our root instance is doing. Let's move onto the components.

ChatList.vue

In your resources/assets/js/components folder, create a file named ChatList.vue with the following contents:

Gist:

[ChatList.vue](#)

From book:

```
<template lang="html">

  <div class="chat-log">

    <chat-message v-for="message in messages"
      v-bind:message="message">
    </chat-message>

    <div class="empty" v-show="messages.length === 0">
      No Messages Yet!
    </div>
  </div>

</template>

<script>

  export default {

    props: [ 'messages' ],

  }
</script>

<style lang="css">

.chat-log .chat-message:nth-child(even) {

  background-color: #f7f7f7;
}

.empty {
  padding: 1rem;
  text-align: center;
}


```

```
</style>
```

Starting with the script, since it's so simple, we see we are bringing in the properties, which we have bound to it from the tag in the chat/index.blade.php view.

And just to reiterate, properties flow down from the root instance to the components, so if we want to use them, we need the following, which we see in our script:

```
export default {  
  props: [ 'messages' ],  
}
```

Next we'll look at the template. You can see we have a component tag for chat-message:

```
<chat-message v-for="message in messages"  
  v-bind:message="message">  
</chat-message>
```

This is exactly like the example we did with the fruit. We iterate through the messages and take the local variable and bind it to message, so we can hand it in to the component. We'll see that in a minute.

Then we have a div that checks to see if we have any messages, and if not, display the No Messages Yet! Message, using v-show:

```
<div class="empty" v-show="messages.length === 0">  
  No Messages Yet!  
</div>
```

So the script and template are actually super simple.

In the style section, we add some styling to alternate the row colors:

```
.chat-list .chat-message:nth-child(even) {  
  background-color: #f7f7f7;  
}
```

Next we'll look at the chat-message component.

ChatMessage.vue

In your resources/assets/js/components folder, create a file named ChatMessage.vue with the following contents:

Gist:

[ChatMessage.vue](#)

From book:

```
<template lang="html">

  <div class="chat-message">

    <p>{{ message.message }}</p>

    <span class="name"> {{ message.user.name }}</span>

    <span class="pull-right">
      <small>
        {{ formattedDate() ? formattedDate() : currentDate() }}
      </small>
    </span>

  </div>

</template>

<script>

  export default {

    props: [ 'message' ],

    methods: {

      formattedDate(){

        if (typeof this.message.created_at === 'undefined' || null) {

          return false;
        }

        let date = this.message.created_at;

        let NowMoment = moment(date);

        return NowMoment.fromNow();
      }
    }
  }
</script>
```

```
},  
  
    currentDate(){  
  
        let now = moment();  
  
        return now.fromNow();  
  
    }  
  
}  
  
</script>  
  
<style lang="css">  
  
.chat-message{  
  
padding: 1rem;  
}  
  
.chat-message > p {  
  
font-size: 1.5rem;  
color: #868889;  
margin-bottom: .5rem;  
}  
  
.name{  
  
font-size: 1.5rem;  
color: #a9c1e8;
```

```
    }  
  
</style>
```

Let's break down the script first. You can see we grab the message property from the parent component, which was chat-list:

```
export default {  
  props: [ 'message' ],
```

Next we have our methods, beginning with the formattedDate method. In this method, we are going to use the Moment.js library for date formatting. We have not installed that, so let's do so now.

Installing Moment.js

If you go to the [Moment.js homepage](#), you will find the instructions for install via you command line and npm:

```
npm install moment --save
```

Once that loads, you need to add the following line to resources/assets/js/bootstrap.js:

```
window.moment = require('moment');
```

Anywhere in the file is fine, I put it under the axios statements.

Ok, now that we have that, we can look at our formattedDate method:

```
formattedDate(){

    if (typeof this.message.created_at === 'undefined'
        || null){

        return false;
    }

    let date = this.message.created_at;

    let NowMoment = moment(date);

    return NowMoment.fromNow();
},
```

We are going to use this in a ternary to determine whether we should show current date time, which is what we do for new messages, or if we are using the created_at date time of messages that have been pulled from the database. If that is not clear, it will be in a moment.

We use our instance of moment to format the date time the way we want it, using the fromNow() method. This format will give us for example:

A few seconds ago

Ok, the next method, currentDate, formats the date if it is a message that has just been posted:

```
currentDate(){

    let now = moment();

    return now.fromNow();
}
```

And that's all there is in our script. Now if we look at our template, we can see how we are using the methods:

```
<p>{{ message.message }}</p>

<span class="name">
  {{ message.user.name }}
</span>

<span class="pull-right">
<small>
  {{ formattedDate() ? formattedDate() : currentDate() }}
</small>
</span>
```

In the `<p>` tag, you see we are getting `message.message`. In this case, `message` is an object, so we have to do `message.message` to get the value.

Next in the `span`, we get the username:

```
<span class="name"> {{ message.user.name }}</span>
```

Then we have our `span` that pulls-right and displays the date time.

```
<span class="pull-right"><small>  
{{ formattedDate() ? formattedDate() : currentDate() }}  
</small></span>
```

Do we have a value for formattedDate? If so, use it, or if not, use the currentDate.

In the style section, feel free to change it however you wish in the display, we will see this in action in a minute, after we finish with chat-create.

ChatCreate.vue

In your resources/assets/js/components folder, create a file named ChatCreate.vue with the following contents:

Gist:

[ChatCreate.vue](#)

From book:

```
<template lang="html">  
  
<div class="chat-create">  
  
  <input type="text"  
         placeholder="Start typing your message..."  
         v-model="messageText"  
         @keyup.enter="sendMessage">  
  
  <button class="btn btn-primary"  
         @click.prevent="sendMessage">  
    Send  
  </button>  
  
</div>  
</template>
```

```
<script>

  export default {

    props: ['currentuser'],

    data() {

      return {

        messageText: ''


      }

    },


    methods: {

      sendMessage(){

        this.$emit('messagecreated', {

          message: this.messageText,

          user: {

            name: this.currentuser

          }
        });

        this.messageText = '';

      }

    }

  }

</script>
```

```
<style lang="css">

.chat-create{

    margin: 2rem;
    display: flex;

}

.chat-create input {

    border: 1px solid;
    flex: 1 auto;
    padding: 1rem;

}

.chat-create button {

    border-radius: 0;
}

</style>
```

Again, let's start with the script. This time the property we need is currentUser:

```
props: [ 'currentuser' ],
```

This comes from the root instance via the binding on the chat-create component tag on chat/index.blade.php. Once again, the property flowing down from the root instance to the component.

Next we have our data, which is a function:

```
data() {  
  return {  
    messageText: ''  
  }  
},
```

We use v-model in the template to bind to that value.

Next we have our sendMessage method:

```
methods: {  
  sendMessage(){  
    this.$emit('messagecreated', {  
      message: this.messageText,  
      user: {  
        name: this.currentuser  
      }  
    });  
    this.messageText = '';  
  }  
}
```

At the end of the message, we set this.messageText to an empty string to clear out the form.

Just two pieces of data need to be sent, this.messageText, which ends up being message.message, and this.current user, which ends up being message.user.

We are using this.\$emit to broadcast our event, which is named messagecreated. We are listening for this our chat-create component tag in chat/index.blade.php:

```
<chat-create v-on:messagecreated="addMessage"
    :currentuser="currentuser"></chat-create>
```

In this case, we have two bindings, currentuser sent down to the component, and the v-on listener for messagecreated, which captures the event coming up from the component and will fire off the addMessage method on the root instance. We already covered that method, but here it is again for reference:

```
addMessage(message) {
    // add to existing messages
    this.messages.push(message);

    // post message to db
    axios.post('/chat-messages', message)
        .then(response => {
            })
        .catch(error => {
            console.log(error);
        });
}
```

Looking at the chat-create component's template, we see it's a text input and a button:

```
<input type="text"
      placeholder="Start typing your message..."'
      v-model="messageText"
      @keyup.enter="sendMessage">

<button class="btn btn-primary"
        @click.prevent="sendMessage">

    Send

</button>
```

We bind the input to the model via v-model, then call the sendMessage method on either keyup.enter or by clicking the button.

And with that, we are ready to compile it down by running from the command line:

```
npm run dev
```

No if you visit:

```
sample-project.com/chat
```

You should see something like:

The screenshot shows a web application interface titled "Sample Project". At the top, there is a navigation bar with links for "Home", "About", "Users", "Content", and a user profile icon labeled "gumby". Below the navigation bar is a list of messages. Each message consists of a text content, a sender name ("gumby"), and a timestamp ("a minute ago"). The messages are:

- perspiciatis ipsum
gumby
- quia odio
gumby
- ab nihil
gumby
- eaque delectus
gumby
- animi officia
gumby
- provident ipsa
gumby
- esse veritatis
gumby

At the bottom of the page is a text input field with placeholder text "Start typing your message..." and a blue "Send" button.

You can see our test messages are being displayed. If you add a message, it will be added to both the list and the database.

Note, you may need to change the timezone in config/app.php if you are not getting the right timezone on your records.

```
'timezone' => 'America/los_Angeles',
```

So this is pretty cool, but what it doesn't do is update other browsers that are watching the room. In fact you can see the badge at the top right displays 0 there is actually one person in the room, you.

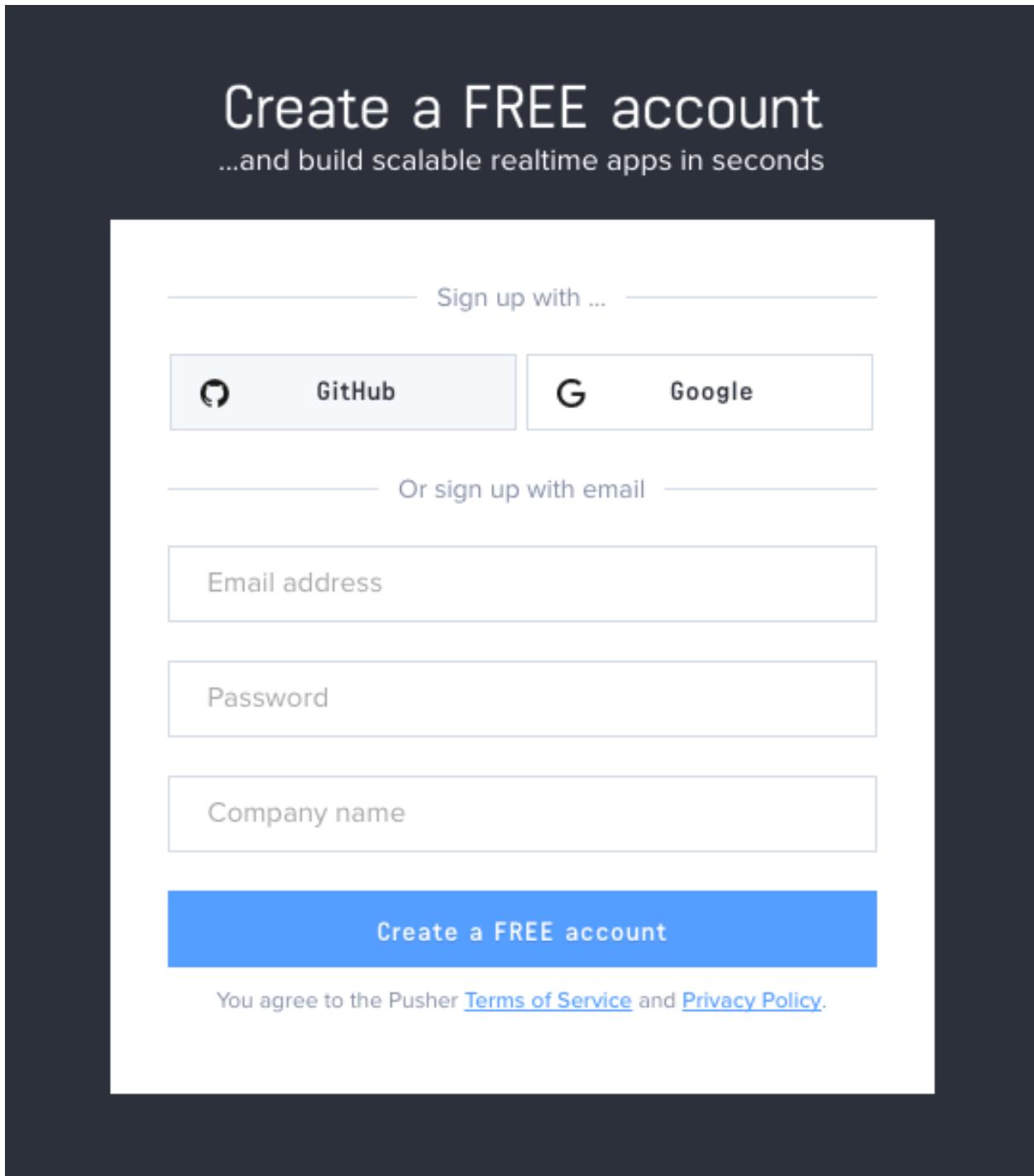
To get the page to update in realtime, we are going to implement Laravel Echo, using Pusher.

Pusher

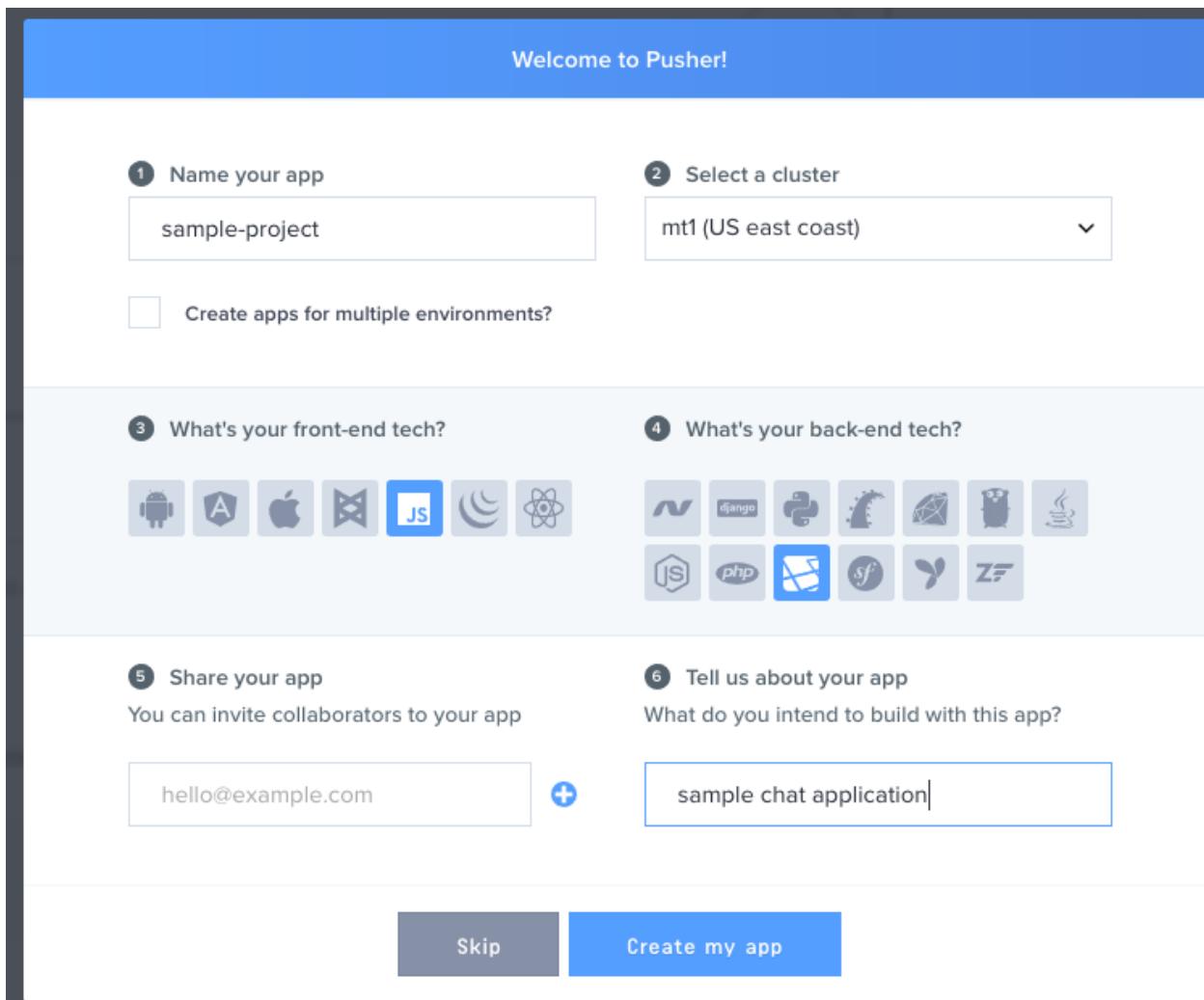
Pusher is a hosted service that makes it super-easy to add real-time data and functionality to web and mobile applications.

I got that directly from the [Laravel Pusher Workshop](#), which you can check out for information on how to use Pusher with Laravel.

To use Pusher, we need to go to [pusher.com](#) and singup for a free account.



I used the one click signup through Github. That brings you to the welcome page:



You can see it's pretty simple. Once you press create my app, you can get your credentials.

You need to paste those into your .env file:

```
PUSHER_APP_ID=your-id  
PUSHER_APP_KEY=your-key  
PUSHER_APP_SECRET=your-secret
```

Also in your .env file is the setting for broadcast driver, you need to set that like so:

```
BROADCAST_DRIVER=pusher
```

And that's it on the Pusher side. Now we need to move onto Laravel Echo.

Laravel Echo

Laravel Echo is included in the standard laravel build, you just need to uncomment the following lines at the bottom of your resources/assets/js/bootstrap.js file:

```
window.Echo = new Echo({
  broadcaster: 'pusher',
  key: 'your-pusher-key'
});
```

Obviously, you'll want to substitute in your actual pusher key as the value for key.

Ok, we're moving right along. Next we will add the following to our root instance of vue in our resources/assets/js/app.js file. The gist will have the entire file for reference, so you know where to place the code, which will go in your created function:

Gist:

[app.js](#)

From book:

```
Echo.join('chatroom')
  .here((users) => {
    this.roomCount = users;
  })
  .joining((user) => {
    this.roomCount.push(user);
  })
});
```

```
})
.leaving((user) => {
  this.roomCount = this.roomCount.filter(u => u != user);
})
.listen('MessagePosted', (e) => {
  this.messages.push({
    message: e.message.message,
    user: e.user
  });
});
```

So let's go over this. The join method of Echo names the room we are joining. In this case, we are telling it 'chatroom,' which is the name we are going to give our room. This is also going to be defined in a new event we are going to create named MessagePosted.

Let's go ahead and create that now by running:

```
php artisan make:event MessagePosted
```

Once you have that in your events folder, let's change the contents of that file.

Gist:

[MessagePosted.php](#)

From book:

```
<?php

namespace App\Events;

use App\Message;
use App\User;
use Illuminate\Broadcasting\Channel;
use Illuminate\Queue\SerializesModels;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Foundation\Events\Dispatchable;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;

class MessagePosted implements ShouldBroadcast
{
    use Dispatchable, InteractsWithSockets, SerializesModels;

    public $message;
    public $user;

    /**
     * Create a new event instance.
     *
     * @return void
     */

    public function __construct(Message $message, User $user)
    {

        $this->message = $message;
        $this->user = $user;

    }

    /**
     * Get the channels the event should broadcast on.
     *
     * @return Channel|array
     */

    public function broadcastOn()
    {
```

```
    return new PresenceChannel('chatroom');

}

}


```

The first thing of note is that we are implementing the ShouldBroadcast contract.

Next you can see that we are setting \$messsage and \$user property through the constructor:

```
public $message;
public $user;

/**
 * Create a new event instance.
 *
 * @return void
 */

public function __construct(Message $message, User $user)
{
    $this->message = $message;
    $this->user = $user;

}
```

When we call the event from our vue instance, we will send along the user and message.

Next, we set the channel:

```
public function broadcastOn()
{
    return new PresenceChannel('chatroom');
}
```

Note we are using PresenceChannel, and that allows us to use the methods on the Vue instance that update the number of people in the room. We will look at that in a moment.

First, let's uncomment the following line in the postMessage method of the ChatController:

```
broadcast(new MessagePosted($message, $user))->toOthers();
```

This fires the event. You could also use the following:

```
event(new MessagePosted($message, $user));
```

But the problem with that, is it will create duplicate messages to the person creating the message. To solve that, we get the broadcast toOthers method.

Ok, let's go back to our root view instance in resources/assets/js/app.js and look again at the Echo methods:

```
Echo.join('chatroom')
  .here((users) => {

    this.roomCount = users;

  })
  .joining((user) => {

    this.roomCount.push(user);

  })
  .leaving((user) => {

    this.roomCount = this.roomCount.filter(u => u != user);

  })
  .listen('MessagePosted', (e) => {

    this.messages.push({

      message: e.message.message,
      user: e.user

    });

  });
});
```

So now we see we have defined ‘chatroom’ in the broadcastOn method of the MessagePosted event, which the .listen method is listening for. Hopefully it’s clear how this is all stitched together.

Let’s look at the methods one by one, starting with the .here method:

```
.here((users) => {

  this.roomCount = users;

})
```

You can see we are simply assigning this.roomCount to the users being handed in from the method, where we hold them in an array on our data model.

Next we have the .joining method:

```
.joining((user) => {  
  this.roomCount.push(user);  
})
```

This announces a new user has joined, so we push that into our roomCount array.

Here we have the .leaving method:

```
.leaving((user) => {  
  this.roomCount = this.roomCount.filter(u => u != user);  
})
```

This announces a user leaving by handing in the user. We run the filter method on roomCount to update the count.

That little piece of code threw me off a little until I realized it was ES6, which allows you to skip the return keyword and the curly braces if it's a single line. In plain javascript, it would look like this:

```
.leaving(function(user){  
  
    this.roomCount = this.roomCount.filter(function(u){  
        return u != user});  
  
})
```

Ok, next we have the listen method:

```
.listen('MessagePosted', (e) => {  
  
    this.messages.push({  
  
        message: e.message.message,  
        user: e.user  
  
    });  
  
});
```

The listen method listens for MessagePosted. It brings in the event result (e), then pushes it into this.messages, where it updates the chat-list component on chat/index.blade.php because we have used v-bind to bind the messages to it.

And Vue's two way model binding will instantly update it for us.

```
<chat-list v-bind:messages="messages"></chat-list>
```

Also note, will we are looking at index.blade.php, we have included the following:

```
<span class="badge pull-right">@{{ roomCount.length }}</span>
```

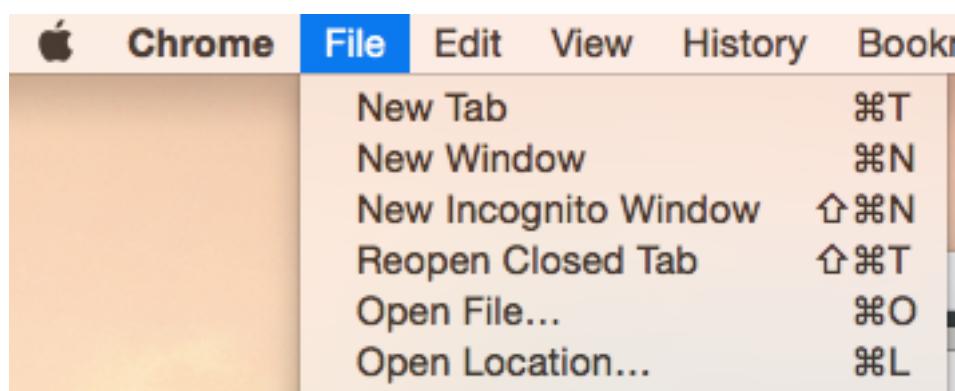
This updates the room count on the page for us.

Ok, we're just about ready to try all this. Let's run the following from the command line:

```
npm run dev
```

And with that you should have real-time updates to your chat room enabled.

Now you will probably want to try this by logging in to two browsers with two different usernames. If you want to do this using chrome only, you need one window to be in incognito mode:

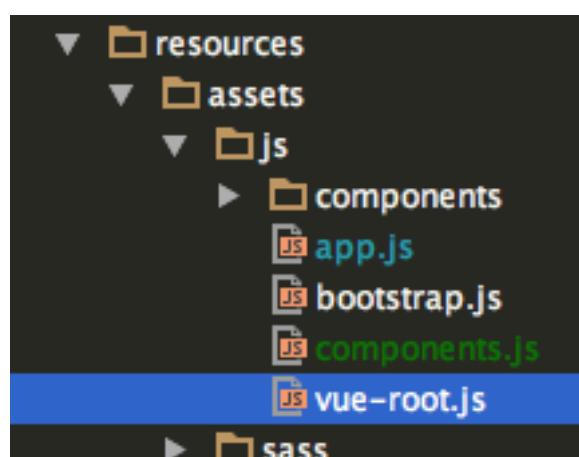


It's the 3rd item on the list.

Ok, play around with that and see if it is all working as planned.

After you've confirmed it's working, let's do one bit of cleanup before we move on.

Let's create a resources/assets/js/vue-root.js file.



Let's chop out the vue instance out of app.js and paste into vue-root.js, so now that file has the following contents:

Gist:

[vue-root](#)

From book:

```
const app = new Vue({  
  
  el: '#app',  
  
  data: {  
  
    messages: [],  
  
    currentuser: '',  
  
    roomCount: []  
  
  },  
  
  methods: {  
  
    addMessage(message) {  
  
      // add to existing messages  
  
      this.messages.push(message);  
  
      axios.post('/chat-messages', message)  
        .then(response => {  
  
          })  
          .catch(error => {  
            console.log(error);  
          });  
  
    },  
  
  },  
});
```

```
created(){

    axios.get('/chat-messages').then(response=> {

        this.messages = response.data;

    });

    axios.get('/username').then(response=> {

        this.currentuser = response.data;

    });

    Echo.join('chatroom')
        .here((users) => {

            this.roomCount = users;

        })
        .joining((user) => {

            this.roomCount.push(user);

        })
        .leaving((user) => {

            this.roomCount =
                this.roomCount.filter(u => u != user);

        })
        .listen('MessagePosted', (e) => {

            this.messages.push({

                message: e.message.message,
                user: e.user

            });

        });

    });

}
```

```
 }  
});  


---


```

Now let's modify resources/assets/js/app.js to the following:

Gist:

[app.js](#)

From book:

```
/**  
 * First we will load all of this project's JavaScript dependencies which  
 * includes Vue and other libraries. It is a great starting point when  
 * building robust, powerful web applications using Vue and Laravel.  
 */  
  
require('./bootstrap');  
  
/**  
 * We will require in our components.js file, which contains our component  
 * files. Putting them in their own file reduces clutter.  
 */  
  
require('./components');  
  
/**  
 * Next, we will create a fresh Vue application instance and attach it to  
 * the page. Then, you may begin adding components to this application  
 * or customize the JavaScript scaffolding to fit your unique needs.  
 * once it gets too big, we will just require it in like other files.  
 */  
  
require('./vue-root');
```

Console Command

If we were running a chat application, one of the concerns would be the number of records that pile up in the database. This could be a huge resource problem, depending on how many messages are being recorded.

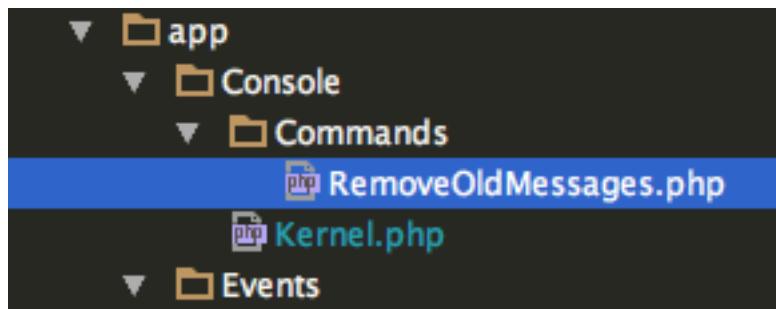
To solve this, we are going to write a console command, which can then be executed by a cron job on the server. I will provide the code for the cron itself, but that will require setup on your server, not your dev environment, and we don't cover that, it's beyond the scope of this book.

So to compensate, we will build in a command line function that will allow you to enter a number for the number of records you want to delete, which will help us make sure it's working. This console command will be available to us via artisan, so we will be able to access it on the command line.

Ok, let's start by running the following from the command line:

```
php artisan make:command RemoveOldMessages
```

That will result in a new Commands folder being created in your Console folder and a stubbed out file for RemoveOldMessages.php, which looks like this:



The contents of the file are:

```
<?php

namespace App\Console\Commands;

use Illuminate\Console\Command;

class RemoveOldMessages extends Command
{
    /**
     * The name and signature of the console command.
     *

```

```
* @var string
*/
protected $signature = 'command:name';

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

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

}

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

}
```

The file has some helpful hints as you can see. We will step through everything one by one, but let me first give you the new file. Let's change RemoveOldMessages.php to the following:

Gist:

RemoveOldMessages.php

From book:

```
<?php

namespace App\Console\Commands;

use Illuminate\Console\Command;
use DB;

class RemoveOldMessages extends Command
{
    /**
     * The name and signature of the console command.
     *
     * @var string
     */
    protected $signature = 'remove:old-chat {count?}';

    /**
     * The console command description.
     *
     * @var string
     */
    protected $description = 'Remove old chat messages from db';

    /**
     * Create a new command instance.
     *
     * @return void
     */
    const MINIMUM_TO_KEEP = 30;

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

```
}

/**
 * Execute the console command.
 *
 * @return mixed
 */
public function handle()
{
    $results = DB::table('messages')->count();

    $safeToDelete = $results >= self::MINIMUM_TO_KEEP ? true : false;

    $count = $this->setCount($results);

    if ($safeToDelete){

        $this->deleteRecords($count);

        return;
    }

    $this->error(
        'Not enough chat messages left to remove '
        . $count . ' old chat messages!'
    );
}

/**
 * @param $results
 * @return array|string
 */
private function setCount($results)
```

```
{  
  
    if ($this->argument('count')) {  
  
        $count = $this->argument('count');  
  
        return $count;  
  
    }  
  
    $count = $results - self::MINIMUM_TO_KEEP;  
  
    return $count;  
  
}  
  
/**  
 * @param $count  
 */  
  
private function deleteRecords($count)  
{  
  
    DB::table('messages')->oldest()->limit($count)->delete();  
  
    $this->info('removed ' . $count . ' old chat messages!');  
  
}  
  
}
```

Ok, so let's start by pointing out that we have added a use statement:

```
use DB;
```

Next we have our signature property:

```
protected $signature = 'remove:old-chat {count?}';
```

This is what defines how the command is called. You can pass in arguments like we did with count, and if you include the question mark, the argument is optional.

You could also set a default value if you wished like so:

```
protected $signature = 'remove:old-chat {count=10}';
```

But that's not what we want here, so make sure it is optional.

Whatever arguments you put in your signature, you will be able to receive in your handle method by using:

```
$this->argument('count')
```

Or

```
$this->arguments()
```

`$this->arguments` brings them in as an array, whereas `$this->argument` is for a single value.

There are more things you can do in the signature, and you should check the [docs](#) for more on that, if you are interested in going further.

Ok, moving on, the next property is our `$description`:

```
protected $description = 'Remove old chat messages from db';
```

This is what will display as a description on the list of artisan commands, we will see that in a minute after we register the command.

Next we've added a constant, so we can set the minimum number of records we want to keep in the db:

```
const MINIMUM_TO_KEEP = 30;
```

We don't want to try to delete records if there are 30 or less records in the db, that is our minimum.

There's nothing notable in the constructor, except that it calls its parent. Then we move on to the handle method, where we've put all our logic.

```
public function handle()
{
    $results = DB::table('messages')->count();

    $safeToDelete = $results >= self::MINIMUM_TO_KEEP ? true : false;

    $count = $this->setCount($results);

    if ($safeToDelete){

        $this->deleteRecords($count);

        return;
    }

    $this->error(
        Not enough chat messages left to remove '
        . $count . ' old chat messages!'
    );
}
```

Let's break it down. First we query the messages table to get a count of the records using count. Then we set \$safeToDelete based on where or not the result of our query is equal to or greater than the MINIMUM_TO_KEEP, which we have set to 30, returning true or false depending on the results of the query.

```
$results = DB::table('messages')->count();

$safeToDelete = $results >= self::MINIMUM_TO_KEEP ? true : false;
```

Next we set the \$count using the setCount method, which takes in the \$results as a parameter:

```
$count = $this->setCount($results);
```

The setCount method, looks like this:

```
private function setCount($results)
{
    if ($this->argument('count')) {

        $count = $this->argument('count');

        return $count;
    }

    $count = $results - self::MINIMUM_TO_KEEP;

    return $count;
}
```

We check the \$this->argument('count') to see if a count has been entered on the command line. If so, we set \$count to that value. If not, we calculate the count by subtracting the minimum number of records we want to keep from the total results that were returned. Then we return \$count.

So back in the handle method, if \$safeToDelete is true, we run:

```
$this->deleteRecords($count);
```

The deleteRecords method is as follows:

```
private function deleteRecords($count)
{
    DB::table('messages')->oldest()->limit($count)->delete();

    $this->info('removed ' . $count . ' old chat messages!');

}
```

We take in the \$count and use it in our query. We are sorting by oldest and are limiting it by the value of \$count.

Then we send some feedback to the browser with the \$this->info() method.

Back in the handle method, we return to finish the block.

If \$safeToDelete evaluates to false, then we skip the if statement and provide feedback to the user:

```
$this->error(
    'Not enough chat messages left to remove '
    . $count . ' old chat messages!'

);
```

We are going to try this in a moment. First, we have to modify our Console/Kernel.php file to the following:

Gist:

[Kernel.php](#)

From book:

```
<?php

namespace App\Console;

use Illuminate\Console\Scheduling\Schedule;
use Illuminate\Foundation\Console\Kernel as ConsoleKernel;

class Kernel extends ConsoleKernel
{
    /**
     * The Artisan commands provided by your application.
     *
     * @var array
     */

    protected $commands = [
        Commands\RemoveOldMessages::class
    ];

    /**
     * Define the application's command schedule.
     *
     * @param \Illuminate\Console\Scheduling\Schedule $schedule
     * @return void
     */
    protected function schedule(Schedule $schedule)
    {
        $schedule->command(Commands\RemoveOldMessages::class)
            ->twiceDaily(1, 13);
    }

    /**
     * Register the Closure based commands for the application.
     *
     * @return void
     */
    protected function commands()
    {
```

```
require base_path('routes/console.php');

}

}
```

We only did 2 changes. First we put our command in the \$commands property:

```
protected $commands = [
    Commands\RemoveOldMessages::class
];
```

That registers the command, so now it will show up in our Artisan list if you run from the command line:

```
php artisan list
```

You will see the following:

| | |
|-------------------|--|
| queue:retry | Retry a failed queue job |
| queue:table | Create a migration for the queue job |
| queue:work | Start processing jobs on the queue |
| remove | |
| remove:old-chat | Remove old chat messages from db |
| route | |
| route:cache | Create a route cache file for faster lookups |
| route:clear | Remove the route cache file |
| route:list | List all registered routes |
| route:cache:clear | Clear the route cache file |

Ok, one last change to the Kernel.php file that we made was in the schedule method, where we specified our class and instructed it to run twice daily, once at 01:00 and once at 13:00:

```
protected function schedule(Schedule $schedule)
{
    $schedule->command(RemoveOldMessages::class)
        ->twiceDaily(1, 13);

}
```

You can see a full list of available scheduling methods in the [docs](#).

So this is not actually going to run by itself. You will need to add the following to your cron settings on your server:

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

As far as I know, MAMP doesn't support that, so we can't run it in the dev environment. However, we can still run the command manually from the command line, for example:

```
php artisan remove:old-chat 2
```

This will remove 2 records. If you don't add a number on the end, it will default to removing all records except for the latest 30.

And that's it! You are ready to maintain your chat application.

Summary

Our goal was to stand up a working chat room and we did it. It's not quite as fancy as slack, but hey, they've got over 600 employees working on their stuff.

Hopefully you've learned more about Laravel Echo, Vue.js and Pusher, and how we can use this combination to quickly stand up realtime browser feedback.

Thanks again for your purchase of the book. Please leave a rating or review at [GoodReads](#), it encourages me to write more and I would really appreciate it. See you soon.

Bill.

Chapter 15: Custom Validators and Vue.js Dependent Dropdown

In the past, when I wanted to do a dependent dropdown list, that is a list where the child is dependent on the parent choice, as it would be in a Category/Subcategory scenario, I was stuck using jquery.

Although I got it to work, I never felt it was a clean solution. So one of my first objectives after finishing the core book was to do one using Vue.js. I'm really happy with the results, so I thought I would share it with the readers of my book.

I always like to learn new things like this in the simplest terms, without cluttering it up with complications, I decided to keep this really simple. We're going to name the models Category and Subcategory, which will avoid confusion.

And while we're doing all this, we're going to create custom a custom validator, so when the form is submitted, we are able to check and see if the subcategory does indeed belong to the category. In your work as a developer, you will no doubt need to create custom validators, so this is a very useful thing to know.

Setting Up The Lesson

We actually need quite a few files to run this example. Since this is ground we have already covered, I'm not going to through each step, instead, I'm going to provide the code and gists where I can, so you can get it set up quickly.

We can start by adding the following routes. I will only show you the new routes, the gist will have the entire file:

Gist: [web.php](#)

From book:

```
Route::get('api/category-data', 'ApiController@categoryData');
Route::get('api/subcategory-data', 'ApiController@subcategoryData');

// Category route

Route::resource('category', 'CategoryController');

// Subcategory route

Route::resource('subcategory', 'SubcategoryController');
```

Category Model

Next run the following from the command line:

```
php artisan make:model Category -m
```

Change the model, Category.php to the following:

Gist:

[Category.php](#)

From book:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Category extends Model
{
    protected $fillable = ['name'];

    public function subcategories()
    {
        return $this->hasMany(Subcategory::class);
    }
}
```

Category Migration

Let's change the contents of the category migration file to the following:

```
<?php

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

class CreateCategoriesTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('categories', function (Blueprint $table) {

            $table->increments('id');
            $table->string('name')->unique();
            $table->timestamps();

        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {

        Schema::dropIfExists('categories');

    }
}
```

Let's run from the command line:

```
php artisan migrate
```

Category Controller

Next let's make the controller, using the command:

```
php artisan make:controller CategoryController
```

Let's change the contents of CategoryController to the following:

Gist:

[CategoryController.php](#)

From book:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Category;
use Illuminate\Support\Facades\Redirect;

class CategoryController extends Controller
{

    public function __construct()
    {
        $this->middleware('auth');

    }

    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */

    public function index()
    {
```

```
    return view('category.index');

}

/**
 * Show the form for creating a new resource.
 *
 * @return \Illuminate\Http\Response
 */

public function create()
{
    return view('category.create');

}

/**
 * Store a newly created resource in storage.
 *
 * @param \Illuminate\Http\Request $request
 * @return \Illuminate\Http\Response
 */
public function store(Request $request)
{
    $this->validate($request, [
        'name' => 'required|unique:categories|string|max:30',
    ]);

    $category = Category::create(['name' => $request->name]);

    $category->save();

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

/**
 * Display the specified resource.
 *
```

```
* @param int $id
* @return \Illuminate\Http\Response
*/
public function show($id)
{
    $category = Category::findOrFail($id);

    return view('category.show', compact('category'));
}

/**
 * Show the form for editing the specified resource.
 *
 * @param int $id
 * @return \Illuminate\Http\Response
 */
public function edit($id)
{
    $category = Category::findOrFail($id);

    return view('category.edit', compact('category'));
}

/**
 * Update the specified resource in storage.
 *
 * @param \Illuminate\Http\Request $request
 * @param int $id
 * @return \Illuminate\Http\Response
 */
public function update(Request $request, $id)
{
    $this->validate($request, [
        'name' =>
            'required|string|max:40|unique:categories,name,' .
            '$id
    ]);
}
```

```
$category = Category::findOrFail($id);

$category->update(['name' => $request->name]);

return Redirect::route('category.show', [
    'category' => $category
]);

}

/**
 * Remove the specified resource from storage.
 *
 * @param int $id
 * @return \Illuminate\Http\Response
 */

public function destroy($id)
{
    Category::destroy($id);

    return Redirect::route('category.index');
}
```

Category Views

Let's start by making a category folder in our resources/views folder.

category/create.blade.php

Gist:

[create.blade.php](#)

From book:

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

@section('title')
    <title>Create a Category</title>

@endsection

@section('content')
    <ol class='breadcrumb'>
        <li><a href='/'>Home</a></li>
        <li><a href='/category'>Categories</a></li>
        <li class='active'>Create</li>
    </ol>

    <h2>Create a New Category</h2>

    <hr/>

    <form class="form"
          role="form"
          method="POST"
          action="{{ url('/category') }}">

        {{ csrf_field() }}

        <!-- Category Name Form Input -->

        <div class="form-group{{ $errors->has('name') ? ' has-error' : '' }}>

            <label class="control-label">Category Name</label>

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

            @if ($errors->has('name'))
                <span class="help-block">
                    <strong>{{ $errors->first('name') }}</strong>
                </span>
            
```

```
@endif

</div>

<div class="form-group">

    <button type="submit"
            class="btn btn-primary btn-lg">

        Create

    </button>

</div>

</form>

@endsection
```

category/edit.blade.php

Gist:

[edit.blade.php](<https://gist.github.com/evercode1/850f6487adcaa7409fb1d28ec50ef179>)

From book:

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

@section('title')

    <title>Edit Category</title>

@endsection

@section('content')
```

```
<ol class='breadcrumb'>
  <li><a href='/'>Home</a></li>
  <li><a href='/category'>Categories</a></li>
  <li><a href='/category/{{ $category->id }}'>
    {{$category->name}}</a></li>
  <li class='active'>Edit</li>
</ol>

<h1>Edit Category</h1>

<hr/>

<form class="form"
      role="form"
      method="POST"
      action="{{ url('/category/'. $category->id) }}"
      {{ method_field('PATCH') }}
      {{ csrf_field() }}>

  <!-- Category Name Form Input -->

  <div class="form-group{{ $errors->has('name') ? ' has-error' : '' }}>

    <label class="control-label">Widget Name</label>

    <input type="text"
           class="form-control"
           name="name"
           value="{{ $category->name }}>

    @if ($errors->has('name'))
      <span class="help-block">
        <strong>{{ $errors->first('name') }}</strong>
      </span>
    
```

```
@endif

</div>

<div class="form-group">

<button type="submit"
        class="btn btn-primary btn-lg">
    Edit
</button>

</div>

</form>

@endsection
```

category/index.blade.php

Gist:

[index.blade.php](#)

From book:

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

@section('title')
    <title>Categories</title>
@endsection

@section('content')
```

```
<ol class='breadcrumb'>
  <li><a href='/'>Home</a></li>
  <li class='active'>Categories</li>
</ol>

<h2>Categories</h2>

<hr/>

<category-grid></category-grid>

<div> <a href="/category/create">

  <button type="button"
    class="btn btn-lg btn-primary">

    Create New

  </button>

</a>
</div>

@endsection
```

category/show.blade.php

Gist:

[show.blade.php](#)

From book:

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

@section('title')
    <title>Category</title>

@endsection

@section('content')
    <ol class='breadcrumb'>
        <li><a href='/'>Home</a></li>
        <li><a href='/category'>Categories</a></li>
        <li><a href='/category/{{ $category->id }}'>
            {{ $category->name }}</a></li>
    </ol>

    <h1>Category Details</h1>

    <hr/>

    <div class="panel panel-default">
        <!-- Table -->
        <table class="table table-striped">
            <thead>
                <tr>
                    <th>Id</th>
                    <th>Name</th>
                    <th>Date Created</th>
                    <th>Edit</th>
                    <th>Delete</th>
                </tr>
            </thead>
            <tbody>
```

```
<tr>

  <td>
    {{ $category->id }}
  </td>
  <td> <a href="/category/{{ $category->id }}/edit">
    {{ $category->name }}
  </a>
  </td>
  <td>
    {{ $category->created_at }}
  </td>
  <td> <a href="/category/{{ $category->id }}/edit">
    <button type="button"
      class="btn btn-default">
      Edit
    </button>
  </a>
  </td>
  <td>
    <div class="form-group">
      <form class="form"
        role="form"
        method="POST"
        action="{{ url('/category/' . $category->id) }}>
```

```
<input type="hidden"
       name="_method"
       value="delete">

{{ csrf_field() }}

<input class="btn btn-danger"
       Onclick="return ConfirmDelete();"
       type="submit"
       value="Delete">

</form>

</div>

</td>

</tr>

</tbody>

</table>

</div>

@endsection

@section('scripts')

<script>

    function ConfirmDelete()
    {

        var x = confirm("Are you sure you want to delete?");
        return x;
    }

</script>

@endsection
```

CategoryGrid

Ok, let's get our Vue component made for our datagrid at resources/assets/js/components:

Gist:

[CategoryGrid.vue](#)

From book:

```
<template>

<div class="row">

  <div class="col-lg-12">

    <form id="search">

      Search

      <input name="query"
             v-model="query"
             @keyup="search(query)">

    </form>

    <div class="pull-right">

      {{ total }} Total Results

    </div>

  <section class="panel">

    <div class="panel-body">

      <table class="table table-bordered table-striped">

        <thead>
```

```
<tr>

  <th v-for="key in gridColumnns"
      @click="sortBy(key)"
      v-bind:class="{active: sortKey == key}">

    {{ key }}

    <span class="arrow"
          v-bind:class="sortOrder > 0 ? 'asc' : 'dsc'">
    </span>

  </th>

<th>

  Actions

</th>

</tr>

</thead>

<tbody>

<tr v-for="row in gridData">

  <td>

    {{row.Id}}


  </td>

  <td>

    <a v-bind:href="'/category/' + row.Id">

      {{row.Name}}


    </a>

  </td>

</tr>
```

```
<td>

{{row.Created} }

</td>

<td>

<a v-bind:href="/category/' + row.Id + '/edit'">

<button type="button"
        class="btn btn-default">

    Edit

</button>

</a>

</td>

</tr>

</tbody>

</table>

</div>

<div class="pull-right">

page {{ current_page }} of {{ last_page }} pages

</div>

</section>

<div class="row">
    <div class="pull-right for-page-button">

        <button @click="getData(go_to_page)"
                class="btn btn-default">

            Go To Page:

        </button>

    </div>
</div>
```

```
</button>

<input v-model="go_to_page"
       class="number-input">
</input>

</div>

<!-- paginate here -->

<ul class="pagination pull-right">

  <li>
    <a @click.prevent="getData(first_page_url)">
      first
    </a>
  </li>

  <li v-if="checkUrlNotNull(prev_page_url)">
    <a @click.prevent="getData(prev_page_url)">
      prev
    </a>
  </li>

  <li v-for="page in pages"
      v-if="page > current_page - 2 && page < current_page + 2"
      v-bind:class="{'active': checkPage(page)}">
    <a @click.prevent="getData(page)">{{ page }}</a>
  </li>

  <li v-if="checkUrlNotNull(next_page_url)">
    <a @click.prevent="getData(next_page_url)">
      next
    </a>
  </li>
</ul>
```

```
</a>

</li>
<li>

    <a @click.prevent="getData(last_page_url)">

        last

    </a>

</li>

</ul>

</div>

</div>

</div>

</template>

<script>

export default {

    mounted: function () {

        this.loadData();

    },

    data: function () {

        return {

            query: '',
            gridColumns: ['Id', 'Name', 'Created'],
            gridData: [],
            total: null,
            next_page_url: null,
        }
    }
}
```

```
    prev_page_url: null,
    last_page: null,
    current_page: null,
    pages: [],
    first_page_url: null,
    last_page_url: null,
    go_to_page: null,
    sortOrder: 1,
    sortKey: ''

}

},
methods: {

  sortBy: function (key){

    this.sortKey = key;
    this.sortOrder = (this.sortOrder == 1) ? -1 : 1;
    this.getData(1);

  },

  search: function(query){

    this.getData(query);

  },

  loadData: function (){

    $.getJSON('api/category-data', function (data) {

      this.gridData = data.data;
      this.total = data.total;
      this.last_page = data.last_page;
      this.next_page_url = data.next_page_url;
      this.prev_page_url = data.prev_page_url;
      this.current_page = data.current_page;
      this.first_page_url = 'api/category-data?page=1';
      this.last_page_url = 'api/category-data?page='
      + this.last_page;

    });

  }

};
```

```
        this.setPageNumbers();

    }.bind(this));
}

setPageNumbers: function(){
    for (var i = 1; i <= this.last_page; i++) {
        this.pages.push(i);
    }
},

getData: function (request){
    let getPage;

    switch (request){

        case this.prev_page_url :
            getPage = this.prev_page_url +
                '&column=' + this.sortKey +
                '&direction=' + this.sortOrder;

            break;

        case this.next_page_url :
            getPage = this.next_page_url +
                '&column=' + this.sortKey +
                '&direction=' + this.sortOrder;

            break;

        case this.first_page_url :
            getPage = this.first_page_url +
                '&column=' + this.sortKey +
                '&direction=' + this.sortOrder;
    }
}
```

```
        break;

    case this.last_page_url :

        getPage = this.last_page_url +
            '&column=' + this.sortKey +
            '&direction=' + this.sortOrder;

        break;

    case this.query :

        getPage = 'api/category-data?' +
            'keyword=' + this.query +
            '&column=' + this.sortKey +
            '&direction=' + this.sortOrder;

        break;

    case this.go_to_page :

        if( this.go_to_page != '' && this.pageInRange()){

            getPage = 'api/category-data?' +
                'page=' + this.go_to_page +
                '&column=' + this.sortKey +
                '&direction=' + this.sortOrder +
                '&keyword=' + this.query;

            this.clearPageNumberInputBox();

        } else {

            alert('Please enter a valid page number');
        }

        break;

    default :

        getPage = 'api/category-data?' +
            'page=' + request +
            '&column=' + this.sortKey +
```

```
        '&direction=' + this.sortOrder +
        '&keyword=' + this.query;

        break;
    }

    if (this.query == '' && getPage != null){

        $.getJSON(getPage, function (data) {

            this.gridData = data.data;
            this.total = data.total;
            this.last_page = data.last_page;
            this.next_page_url = data.next_page_url;
            this.prev_page_url = data.prev_page_url;
            this.current_page = data.current_page;

        }).bind(this));

    } else {

        if (getPage != null){

            $.getJSON(getPage, function (data) {

                this.gridData = data.data;
                this.total = data.total;
                this.last_page = data.last_page;
                this.next_page_url =
                    (data.next_page_url == null) ? null :
                    data.next_page_url + '&keyword='
                    +this.query;
                this.prev_page_url =
                    (data.prev_page_url == null) ? null :
                    data.prev_page_url + '&keyword=' +this.query;
                this.first_page_url =
                    'api/category-data?page=1&keyword='
                    +this.query;
                this.last_page_url = 'api/category-data?page='
                    + this.last_page + '&keyword=' +this.query;
                this.current_page = data.current_page;
                this.resetPageNumbers();

            }).bind(this));

        }
    }
}
```

```
        }

    }

    ,
checkPage: function(page){

    return page == this.current_page;

},
resetPageNumbers: function(){

    this.pages = [];

    for (var i = 1; i <= this.last_page; i++) {

        this.pages.push(i);

    }

},
checkUrlNotNull: function(url){

    return url != null;

},
clearPageNumberInputBox: function(){

    return this.go_to_page = '';

},
pageInRange: function(){

    return this.go_to_page <= parseInt(this.last_page);

}
}
```

```
    }  
  
</script>
```

Definitely use the gist on that one.

components.js

Let's add the call to the category component in our resources/assets/js/components.js file:

```
Vue.component('category-grid',  
  require('./components/CategoryGrid.vue'));
```

Should be one line, not two.

Add categoryData method to ApiController.php

Gist:

[ApiController.php](#)

From book:

```
public function categoryData(Request $request)  
{  
  
    return GridQuery::sendData($request, new CategoryQuery);  
}
```

Don't forget the use statement at the top:

```
use App\Queries\GridQueries\CategoryQuery;
```

CategoryQuery.php

In app/Queries/GridQueries, let's create CategoryQuery.php:

Gist:

[CategoryQuery.php](#)

From book:

```
<?php

namespace App\Queries\GridQueries;
use DB;
use App\Queries\GridQueries\Contracts\DataQuery;

class CategoryQuery implements DataQuery
{

    public function data($column, $direction)
    {

        $rows = DB::table('categories')
            ->select('id as Id',
                      'name as Name',
                      DB::raw('DATE_FORMAT(created_at,
                        "%m-%d-%Y") as Created'))
            ->orderBy($column, $direction)
            ->paginate(10);

        return $rows;
    }

    public function filteredData($column, $direction, $keyword)
    {

```

```

$rows = DB::table('categories')
    ->select('id as Id',
              'name as Name',
              DB::raw('DATE_FORMAT(created_at,
                      "%m-%d-%Y") as Created'))
    ->where('name', 'like', '%' . $keyword . '%')
    ->orderBy($column, $direction)
    ->paginate(10);

return $rows;

}

}

```

Category Factory

Let's add the following to our database/factories/ModelFactory.php file:

Gist:

[category factory method](#)

From book:

```

$factory->define(App\Category::class, function (Faker\Generator $faker) {

    return [
        'name' => $faker->unique()->word,
    ];
});
```

And that's it for Category. Now we have to do Subcategory.

Subcategory

Subcategory Model

Let's run the following from the command line:

```
php artisan make:model Subcategory -m
```

Next, let's modify Subcategory.php to the following:

Gist:

[Subcategory.php](#)

From book:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Subcategory extends Model
{
    protected $fillable = ['name', 'category_id'];

    public function category()
    {
        return $this->belongsTo(Category::class);
    }
}
```

Subcategory Migration

Next, let's change the subcategory migration file to the following:

Gist:

[subcategory migration](#)

From book:

```
<?php

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

class CreateSubcategoriesTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('subcategories', function (Blueprint $table) {

            $table->increments('id');
            $table->string('name')->unique();
            $table->integer('category_id')->unsigned();
            $table->timestamps();

        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
```

```
Schema::dropIfExists('subcategories');

}

}
```

Let's go ahead and migrate:

```
php artisan migrate
```

Subcategory Controller

Gist:

[SubcategoryController.php](<https://gist.github.com/evercode1/9a5b7b2e1c5ca5aa2310736239bd078e>)

From book:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Subcategory;
use App\Category;
use Illuminate\Support\Facades\Redirect;

class SubcategoryController extends Controller
{

    public function __construct()
    {
        $this->middleware('auth');
```

```
}

/**
 * Display a listing of the resource.
 *
 * @return \Illuminate\Http\Response
 */

public function index()
{
    return view('subcategory.index');

}

/**
 * Show the form for creating a new resource.
 *
 * @return \Illuminate\Http\Response
 */

public function create()
{
    $categories = Category::orderBy('name', 'asc')->get();

    return view('subcategory.create', compact('categories'));
}

/**
 * Store a newly created resource in storage.
 *
 * @param \Illuminate\Http\Request $request
 * @return \Illuminate\Http\Response
 */

public function store(Request $request)
{
    $count = Category::count();

    $this->validate($request, [
        'category_name' => 'required|exists:categories,name'
    ]);

    if ($count >= 10) {
        $request->session()->flash('error', 'Category limit reached');
        return redirect()->back();
    }

    $category = new Category();
    $category->name = $request->category_name;
    $category->description = $request->category_description;
    $category->parent_id = $request->parent_id;
    $category->order = $request->order;
    $category->status = $request->status;
    $category->created_at = now();
    $category->updated_at = now();

    $category->save();

    $request->session()->flash('success', 'Category created successfully');
    return redirect()->route('subcategory.index');
}
```

```
'name' => 'required|unique:subcategories|string|max:30',
'category_id' => "required|numeric|digits_between:1,$count"
]);

$subcategory = Subcategory::create([
    'name' => $request->name,
    'category_id' => $request->category_id
]);

$subcategory->save();

return Redirect::route('subcategory.index');

}

/**
 * Display the specified resource.
 *
 * @param int $id
 * @return \Illuminate\Http\Response
 */
public function show($id)
{
    $subcategory = Subcategory::findOrFail($id);

    $category = $subcategory->category->name;

    return view('subcategory.show',
        compact('subcategory', 'category'));
}

/**
 * Show the form for editing the specified resource.
 *
 * @param int $id
 * @return \Illuminate\Http\Response
 */
public function edit($id)
```

```
{  
    $subcategory = Subcategory::findOrFail($id);  
  
    $categories = Category::orderBy('name', 'asc')->get();  
  
    $oldValue = $subcategory->category->name;  
  
    $oldId = $subcategory->category->id;  
  
    return view('subcategory.edit', compact(  
        'subcategory',  
        'categories',  
        'oldValue',  
        'oldId'  
    ));  
}  
  
/**  
 * Update the specified resource in storage.  
 *  
 * @param \Illuminate\Http\Request $request  
 * @param int $id  
 * @return \Illuminate\Http\Response  
 */  
  
public function update(Request $request, $id)  
{  
  
    $count = Category::count();  
  
    $this->validate($request, [  
  
        'name' =>  
        'required|string|max:40|unique:subcategories,name,' . $id,  
        'category_id' => "required|numeric|digits_between:1,$count"  
    ]);  
  
    $subcategory = Subcategory::findOrFail($id);  
  
    $subcategory->update([
```

```

    'name' => $request->name,
    'category_id' => $request->category_id
]);

$category = $subcategory->category->name;

return Redirect::route('subcategory.show', [
    'subcategory' => $subcategory,
    'category' => $category
]);

}

/**
 * Remove the specified resource from storage.
 *
 * @param int $id
 * @return \Illuminate\Http\Response
 */
public function destroy($id)
{
    Subcategory::destroy($id);

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

```

Subcategory Views

Let's make a view folder named subcategory and put the following blade files in it.

subcategory/create.blade.php

Gist:

[create.blade.php](#)

From book:

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

@section('title')
    <title>Create a Subcategory</title>

@endsection

@section('content')
    <ol class='breadcrumb'>
        <li><a href='/'>Home</a></li>
        <li><a href='/subcategory'>Subcategories</a></li>
        <li class='active'>Create</li>
    </ol>

    <h2>Create a New Subcategory</h2>

    <hr/>

    <form class="form"
          role="form"
          method="POST"
          action="{{ url('/category') }}">

        {{ csrf_field() }}

        <!-- Subcategory Name Form Input -->

        <div class="form-group{{ $errors->has('name') ? ' has-error' : '' }}>

            <label class="control-label">
                Subcategory Name
            </label>

            <input type="text"
```

```
        class="form-control"
        name="name"
        value="{{ old('name') }}"

@if ($errors->has('name'))

<span class="help-block">

<strong>{{ $errors->first('name') }}</strong>

</span>

@endif

</div>

<!-- Category Select Form -->

<div class="

form-group{{ $errors->has('category_id') ? ' has-error' : '' }}>

">

<label for="category_id">

Category Name:

</label>

<select class="form-control"
        name="category_id">

<option value="">Please Choose One</option>

@foreach($categories as $category)

<option value="{{ $category->id }}>
    {{ $category->name }}
</option>

@endforeach

```

```
</select>

@if ($errors->has('category_id'))

    <span class="help-block">
        <strong>{{ $errors->first('category_id') }}</strong>
    </span>

@endif

</div>

<div class="form-group">

    <button type="submit" class="btn btn-primary btn-lg">
        Create
    </button>
</div>

</form>

@endsection
```

subcategory/edit.blade.php

Gist:

[edit.blade.php](#)

From book:

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

@section('title')

<title>Edit Subcategory</title>

@endsection

@section('content')

<ol class='breadcrumb'>
    <li><a href='/'>Home</a></li>
    <li><a href='/subcategory'>Subcategories</a></li>
    <li><a href='/subcategory/{{ $subcategory->id }}'>
        {{$subcategory->name}}</a></li>
    <li class='active'>Edit</li>
</ol>

<h1>Edit Subcategory</h1>

<hr/>

<form class="form"
      role="form"
      method="POST"
      action="{{ url('/subcategory/' . $subcategory->id) }}"
      {{ method_field('PATCH') }}
      {{ csrf_field() }}>

<div class="form-group{{ $errors->has('name') ? ' has-error' : '' }}>

    <label class="control-label">

        Subcategory Name

    </label>

    <input type="text"
          class="form-control"
          name="name"
```

```
        value="{{ $subcategory->name }}"

@if ($errors->has('name'))

<span class="help-block">

<strong>{{ $errors->first('name') }}</strong>

</span>

@endif

</div>

<!-- Category Form Select -->

<div class="

form-group{{ $errors->has('category_id') ? ' has-error' : '' }}" >

<label for="category_id">

Category Name:

</label>

<select class="form-control"
       name="category_id">

<option value="{{ $oldId }}>

{{ $oldValue }}
```

```
@endforeach

</select>

@if ($errors->has('category_id'))

<span class="help-block">

<strong>

{{ $errors->first('category_id') }}

</strong>

</span>

@endif

</div>

<div class="form-group">

<button type="submit"
        class="btn btn-primary btn-lg">

    Edit

</button>

</div>

</form>

@endsection
```

subcategory/index.blade.php

Gist:

index.blade.php

From book: _____

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

@section('title')
    <title>Subcategories</title>

@endsection

@section('content')
    <ol class='breadcrumb'>
        <li><a href='/'>Home</a></li>
        <li class='active'>Subcategories</li>
    </ol>

    <h2>Subcategories</h2>

    <hr/>

    <subcategory-grid></subcategory-grid>

    <div> <a href="subcategory/create">

        <button type="button"
            class="btn btn-lg btn-primary">

            Create New

        </button>

    </a>

    </div>

@endsection
```

subcategory/show.blade.php

Gist:

show.blade.php

From book:

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

@section('title')
    <title>Subcategory</title>

@endsection

@section('content')
    <ol class='breadcrumb'>
        <li><a href='/'>Home</a></li>
        <li><a href='/subcategory'>Subcategories</a></li>
        <li><a href='/subcategory/{{ $subcategory->id }}'>
            {{ $subcategory->name }}</a></li>
    </ol>

    <h1>Subcategory Details</h1>

    <hr/>

    <div class="panel panel-default">
        <!-- Table -->
        <table class="table table-striped">
            <thead>
                <tr>
                    <th>Id</th>
                    <th>Name</th>
                    <th>Category</th>
                    <th>Date Created</th>
                    <th>Edit</th>
                    <th>Delete</th>
                </tr>
            </thead>
```

```
</tr>

</thead>

<tbody>

<tr>
  <td>{{ $subcategory->id }}</td>

  <td><a href="/subcategory/{{ $subcategory->id }}/edit">
    {{ $subcategory->name }}
  </a></td>

  <td>{{ $category }}</td>

  <td>{{ $subcategory->created_at }}</td>

  <td> <a href="/subcategory/{{ $subcategory->id }}/edit">
    <button type="button"
      class="btn btn-default">
      Edit
    </button></a></td>

  <td>

    <div class="form-group">

      <form class="form"
        role="form"
        method="POST"
        action="{{ url('/subcategory/'. $subcategory->id) }}">

        <input type="hidden"
          name="_method"
          value="delete">

        {{ csrf_field() }}
    
```

```
<input class="btn btn-danger"
      onclick="return ConfirmDelete();"
      type="submit"
      value="Delete">

</form>

</div>

</td>

</tr>

</tbody>

</table>

</div>

@endsection

@section('scripts')

<script>

function ConfirmDelete()
{
    var x = confirm("Are you sure you want to delete?");
    return x;
}

</script>

@endsection
```

SubcategoryGrid.vue

Gist:

[SubcategoryGrid.vue](#)

From book:

```
<template>

<div class="row">
  <div class="col-lg-12">
    <form id="search">
      Search
      <input name="query"
             v-model="query" @keyup="search(query)">
    </form>
    <div class="pull-right">
      {{ total }} Total Results
    </div>
  <section class="panel">
    <div class="panel-body">
      <table class="table table-bordered table-striped">
        <thead>
          <tr>
            <th v-for="key in gridColumns"
                @click="sortBy(key)"
                v-bind:class="{active: sortKey == key}">
              {{ key }}
            <span class="arrow"
                  v-bind:class="sortOrder > 0 ? 'asc' : 'dsc'">
            </span>
          </th>
        </thead>
        <tbody>
          <tr v-for="item in items">
            <td>{{ item.id }}</td>
            <td>{{ item.name }}</td>
            <td>{{ item.type }}</td>
            <td>{{ item.value }}</td>
            <td>{{ item.date }}</td>
            <td>{{ item.status }}</td>
            <td>{{ item.actions }}</td>
          </tr>
        </tbody>
      </table>
    </div>
  </section>
</div>
```

```
</th>

<th>Actions</th>

</tr>
</thead>

<tbody>

<tr v-for="row in gridData">

    <td>
        {{row.Id}}
    </td>

    <td>
        <a v-bind:href="'/subcategory/' + row.Id">
            {{row.Name}}
        </a>
    </td>

    <td>
        <a v-bind:href="'/category/' + row.CategoryId">
            {{row.Category}}
        </a>
    </td>

    <td>
        {{row.Created}}
    </td>


```

```
<td>

  <a v-bind:href="' /subcategory/' +row.Id +'/edit'">

    <button type="button"
      class="btn btn-default">

      Edit

    </button>

  </a>

</td>

</tr>

</tbody>

</table>

</div>

<div class="pull-right">

  page {{ current_page }} of {{ last_page }} pages

</div>

</section>

<div class="row">

  <div class="pull-right for-page-button">

    <button @click="getData(go_to_page)"
      class="btn btn-default">

      Go To Page:

    </button>

    <input v-model="go_to_page"
      class="number-input">

  </div>

</div>
```

```
</input>

</div>

<!-- paginate here -->

<ul class="pagination pull-right">

<li>

<a @click.prevent="getData(first_page_url)">

    first

</a>

</li>

<li v-if="checkUrlNotNull(prev_page_url)">

    <a @click.prevent="getData(prev_page_url)">

        prev

    </a>

</li>

<li v-for="page in pages"
    v-if="page > current_page - 2 && page < current_page + 2"
    v-bind:class="{ 'active': checkPage(page) }">

    <a @click.prevent="getData(page)">

        {{ page }}

    </a>

</li>

<li v-if="checkUrlNotNull(next_page_url)">

    <a @click.prevent="getData(next_page_url)">
```

```
    next

    </a>

</li>

<li>

    <a @click.prevent="getData(last_page_url)">

        last

    </a>

</li>

</ul>

</div>

</div>

</template>

<script>

export default {

    mounted: function () {

        this.loadData();

    },

    data: function () {

        return {

            query: '',
            gridColumns: ['Id', 'Name', 'Category', 'Created'],
            gridData: [],

```

```
        total: null,
        next_page_url: null,
        prev_page_url: null,
        last_page: null,
        current_page: null,
        pages: [],
        first_page_url: null,
        last_page_url: null,
        go_to_page: null,
        sortOrder: 1,
        sortKey: ''
    }

    },
methods: {

    sortBy: function (key){

        this.sortKey = key;
        this.sortOrder = (this.sortOrder == 1) ? -1 : 1;
        this.getData(1);

    },
    search: function(query){

        this.getData(query);

    },
    loadData: function (){

        $.getJSON('api/subcategory-data', function (data) {

            this.gridData = data.data;
            this.total = data.total;
            this.last_page = data.last_page;
            this.next_page_url = data.next_page_url;
            this.prev_page_url = data.prev_page_url;
            this.current_page = data.current_page;
            this.first_page_url =

```

```
'api/subcategory-data?page=1';
this.last_page_url =
'api/subcategory-data?page=' +
this.last_page;
this.setPageNumbers();

}.bind(this));

} ,

setPageNumbers: function(){
for (var i = 1; i <= this.last_page; i++) {
    this.pages.push(i);
}

} ,

getData: function (request){
let getPage;

switch (request){

case this.prev_page_url :
    getPage = this.prev_page_url +
        '&column=' + this.sortKey +
        '&direction=' + this.sortOrder;

    break;

case this.next_page_url :
    getPage = this.next_page_url +
        '&column=' + this.sortKey +
        '&direction=' + this.sortOrder;

    break;
}
```

```
case this.first_page_url :  
  
    getPage = this.first_page_url +  
        '&column=' + this.sortKey +  
        '&direction=' + this.sortOrder;  
  
    break;  
  
case this.last_page_url :  
  
    getPage = this.last_page_url +  
        '&column=' + this.sortKey +  
        '&direction=' + this.sortOrder;  
  
    break;  
  
case this.query :  
  
    getPage = 'api/subcategory-data?' +  
        'keyword=' + this.query +  
        '&column=' + this.sortKey +  
        '&direction=' + this.sortOrder;  
  
    break;  
  
case this.go_to_page :  
  
    if( this.go_to_page != '' && this.pageInRange()) {  
  
        getPage = 'api/subcategory-data?' +  
            'page=' + this.go_to_page +  
            '&column=' + this.sortKey +  
            '&direction=' + this.sortOrder +  
            '&keyword=' + this.query;  
  
        this.clearPageNumberInputBox();  
  
    } else {  
  
        alert('Please enter a valid page number');  
    }  
  
    break;
```

```
    default :  
  
        getPage = 'api/subcategory-data?' +  
            'page=' + request +  
            '&column=' + this.sortKey +  
            '&direction=' + this.sortOrder +  
            '&keyword=' + this.query;  
  
        break;  
    }  
  
    if (this.query == '' && getPage != null){  
  
        $.getJSON(getPage, function (data) {  
  
            this.gridData = data.data;  
            this.total = data.total;  
            this.last_page = data.last_page;  
            this.next_page_url = data.next_page_url;  
            this.prev_page_url = data.prev_page_url;  
            this.current_page = data.current_page;  
  
            }.bind(this));  
  
    } else {  
  
        if (getPage != null){  
  
            $.getJSON(getPage, function (data) {  
  
                this.gridData = data.data;  
                this.total = data.total;  
                this.last_page = data.last_page;  
                this.next_page_url = (data.next_page_url == null) ?  
                    null : data.next_page_url + '&keyword=' + this.query;  
                this.prev_page_url = (data.prev_page_url == null) ?  
                    null : data.prev_page_url + '&keyword=' + this.query;  
                this.first_page_url =  
                    'api/subcategory-data?page=1&keyword=' + this.query;  
                this.last_page_url =  
                    'api/subcategory-data?page=' +  
                    this.last_page + '&keyword=' + this.query;  
                this.current_page = data.current_page;  
                this.resetPageNumbers();  
            }.bind(this));  
        }  
    }  
}
```

```
        } .bind(this));
    }

},
checkPage: function(page){
    return page == this.current_page;
},
resetPageNumbers: function(){
    this.pages = [];
    for (var i = 1; i <= this.last_page; i++) {
        this.pages.push(i);
    }
},
checkUrlNotNull: function(url){
    return url != null;
},
clearPageNumberInputBox: function(){
    return this.go_to_page = '';
},
pageInRange: function(){
    return this.go_to_page <= parseInt(this.last_page);
}
```

```
        }
    }

</script>
```

components.js

Gist:

[components.js](#)

From book:

```
Vue.component('subcategory-grid',
require('./components/SubcategoryGrid.vue'));
```

ApiController

Add the following method to the ApiController.php file.

Gist:

[ApiController.php](#)

From book:

```
public function subcategoryData(Request $request)
{
    return GridQuery::sendData($request, new SubcategoryQuery);
}
```

Make sure you have the use statement at the top of the file:

```
use App\Queries\GridQueries\SubcategoryQuery;
```

SubcategoryQuery.php

Gist:

[SubcategoryQuery.php](#)

From book:

```
<?php
```

```
namespace App\Queries\GridQueries;
use DB;
use App\Queries\GridQueries\Contracts\DataQuery;

class SubcategoryQuery implements DataQuery
{

    public function data($column, $direction)
    {

        $rows = DB::table('subcategories')
            ->select('subcategories.id as Id',
                      'subcategories.name as Name',
                      'categories.name as Category',
                      'categories.id as CategoryId',
                      DB::raw('DATE_FORMAT(subcategories.created_at,
                                         "%m-%d-%Y") as Created'))
            ->leftJoin('categories',
                        'category_id', '=', 'categories.id')
            ->orderBy($column, $direction)
            ->paginate(10);

        return $rows;
    }

    public function filteredData($column, $direction, $keyword)
```

```

{
    $rows = DB::table('subcategories')
        ->select('subcategories.id as Id',
                  'subcategories.name as Name',
                  'categories.name as Category',
                  'categories.id as CategoryId',
                  DB::raw('DATE_FORMAT(subcategories.created_at,
                      "%m-%d-%Y") as Created'))
        ->leftJoin('categories',
                  'category_id', '=', 'categories.id')
        ->where('subcategories.name',
                  'like', '%' . $keyword . '%')
        ->orWhere('categories.name',
                  'like', '%' . $keyword . '%')
        ->orderBy($column, $direction)
        ->paginate(10);

    return $rows;
}

}

```

Ok, now we need to run from the command line:

```
npm run dev
```

Seed data

We need some seed data to see if it's all working. We made some factory methods for quick seeding, which will quickly populate our tables.

So let's run tinker:

```
php artisan tinker
```

Then let's populate 10 records into category:

```
factory('App\Category', 10)->create()
```

Next let's do 20 records for subcategory:

```
factory('App\Subcategory', 20)->create()
```

And that should result in the datagrids you are used to seeing. That was approximately 44 pages of setup for this lesson, and we're not done yet.

The good news is that now we have full parent/child models to work with, so when we build our custom validation methods, we will be running actual queries to determine whether validation passes or fails.

Right now, you have a parent child relationship named Category and Subcategory. Quite often in a project, you can have those names for the models. Other times you will have things like make and model for cars, for example, where the model is dependent on the make.

In our case, we are sticking with Category and Subcategory, so it's always crystal clear that Subcategory is the child and belongs to Category.

One issue though, is that the latin name seed data is not very user friendly. So what I did in my project was edit the Category records manually, so that they each category represented a programming language or a framework or a programming subject. The result looks like this:

| Id | Name | Created | Actions |
|----|-------------------|------------|---------|
| 1 | Laravel | 02-09-2017 | Edit |
| 2 | PHP | 02-09-2017 | Edit |
| 3 | Javascript | 02-09-2017 | Edit |
| 4 | Vue | 02-09-2017 | Edit |
| 5 | Mix | 02-09-2017 | Edit |
| 6 | Echo | 02-09-2017 | Edit |
| 7 | Dynamic Facades | 02-09-2017 | Edit |
| 8 | Pusher | 02-09-2017 | Edit |
| 9 | Service Providers | 02-09-2017 | Edit |
| 10 | Vue Components | 02-09-2017 | Edit |

Next, I changed my subcategories to relate to the subject as if they represented a lesson or tutorial that related to the category. It looks like so:

| ID | Name | Category | Created | Actions |
|----|------------------|------------|------------|-----------------------|
| 1 | blade | Laravel | 02-09-2017 | <button>Edit</button> |
| 2 | classes | Javascript | 02-09-2017 | <button>Edit</button> |
| 3 | migrations | Laravel | 02-09-2017 | <button>Edit</button> |
| 4 | const | Javascript | 02-09-2017 | <button>Edit</button> |
| 5 | let | Javascript | 02-09-2017 | <button>Edit</button> |
| 6 | mounted | Vue | 02-09-2017 | <button>Edit</button> |
| 7 | image management | Laravel | 02-09-2017 | <button>Edit</button> |
| 8 | variables | PHP | 02-09-2017 | <button>Edit</button> |
| 9 | validation rules | Laravel | 02-09-2017 | <button>Edit</button> |
| 10 | public method | PHP | 02-09-2017 | <button>Edit</button> |

Hopefully the image is clear enough for you to get the idea.

This is a little friendlier seed data and will make it easier for us to know if the dependent dropdown list we create for subcategory is limited to only the categories to which it belongs, which we will only see after the category has been selected.

In terms of setup, we need to create one more model, the one that will use category and subcategory. We'll step through basic setup quickly as we did before.

Don't be alarmed if you notice some missing attributes, here and there. I'm going to skip over including category_id and subcategory_id in the controller and the views, so that we can walk through those parts together.

Route

Add the following routes to routes/web.php:

```
Route::get('lesson/create',
'LessonController@create')->name('lesson.create');

Route::get('lesson/{lesson}-{slug?}',
'LessonController@show')->name('lesson.show');

Route::resource('lesson',
'LessonController', ['except' => ['show', 'create']]);

```

Lesson Model

```
php artisan make:model Lesson -m
```

Lesson.php

Let's change Lesson.php to the following:

Gist:

[Lesson.php](#)

From book:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Lesson extends Model
{
    protected $fillable = [
        'name',
        'slug',
        'category_id',
        'subcategory_id'];
}


```

Next, we'll work on the migration:

Lesson Migration

Gist:

[Lesson Migration](#)

From book:

```
<?php

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

class CreateLessonsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
}
```

```
public function up()
{
    Schema::create('lessons', function (Blueprint $table) {

        $table->increments('id');
        $table->string('name')->unique();
        $table->string('slug')->unique();
        $table->integer('category_id')->unsigned();
        $table->integer('subcategory_id')->unsigned();
        $table->timestamps();

    });
}

/**
 * Reverse the migrations.
 *
 * @return void
 */
public function down()
{
    Schema::dropIfExists('lessons');

}
}
```

LessonController

Let's create a controller from the command line:

```
php artisan make:controller LessonController
```

LessonController.php

Let's change it to the following:

Gist:

[LessonController.php](#)

From book:

```
<?php

namespace App\Http\Controllers;

use App\Http\Auth\Traits\OwnsRecord;
use Illuminate\Http\Request;
use App\Lesson;
use Redirect;
use Illuminate\Support\Facades\Auth;
use App\Exceptions\UnauthorizedException;

class LessonController extends Controller
{
    use OwnsRecord;

    public function __construct()
    {
        $this->middleware(['auth', 'admin']);
    }

    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */

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

```
/**  
 * Show the form for creating a new resource.  
 *  
 * @return \Illuminate\Http\Response  
 */  
  
public function create()  
{  
  
    return view('lesson.create');  
  
}  
  
/**  
 * Store a newly created resource in storage.  
 *  
 * @param \Illuminate\Http\Request $request  
 * @return \Illuminate\Http\Response  
 */  
  
public function store(Request $request)  
{  
  
    $this->validate($request, [  
  
        'name' => 'required|unique:lessons|string|max:30',  
  
    ]);  
  
    $slug = str_slug($request->name, "-");  
  
    $lesson = Lesson::create([  
  
        'name' => $request->name,  
        'slug' => $slug  
  
    ]);  
  
    $lesson->save();  
  
    alert()->success('Congrats!', 'You made a Lesson');  
  
    return Redirect::route('lesson.index');
```

```
}

/**
 * Display the specified resource.
 *
 * @param int $id
 * @return \Illuminate\Http\Response
 */

public function show(Lesson $lesson, $slug = '')
{
    if ($lesson->slug !== $slug) {

        return Redirect::route('lesson.show', [
            'id' => $lesson->id,
            'slug' => $lesson->slug
        ], 301);
    }

    return view('lesson.show', compact('lesson'));
}

/**
 * Show the form for editing the specified resource.
 *
 * @param int $id
 * @return \Illuminate\Http\Response
 */

public function edit(Lesson $lesson)
{
    if ( ! $this->adminOrCurrentUserOwns($lesson)) {

        throw new UnauthorizedException;
    }

    return view('lesson.edit', compact('lesson'));
}
```

```
/**  
 * Update the specified resource in storage.  
 *  
 * @param \Illuminate\Http\Request $request  
 * @param int $id  
 * @return \Illuminate\Http\Response  
 */  
  
public function update(Request $request, Lesson $lesson)  
{  
  
    $this->validate($request, [  
  
        'name' =>  
        'required|string|max:40|unique:widgets,name,'  
        . $lesson->id  
  
    ]);  
  
    if ( ! $this->adminOrCurrentUserOwns($lesson)){  
  
        throw new UnauthorizedException;  
  
    }  
  
    $slug = str_slug($request->name, "-");  
  
    $lesson->update([  
  
        'name' => $request->name,  
        'slug' => $slug  
  
    ]);  
  
    alert()->success('Congrats!', 'You updated a lesson');  
  
    return Redirect::route('lesson.show', [  
  
        'lesson' => $lesson,  
        'slug' => $slug  
  
    ]);  
}
```

```
}

/**
 * Remove the specified resource from storage.
 *
 * @param int $id
 * @return \Illuminate\Http\Response
 */

public function destroy($id)
{
    Lesson::destroy($id);

    alert()->overlay('Attention!',
        'You deleted a lesson', 'error');

    return Redirect::route('lesson.index');
}
```

Lesson Views

Let's make a lesson folder in our views folder.

lesson/create

Gist:

[create.blade.php](#)

From book:

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

@section('title')
    <title>Create a Lesson</title>

@endsection

@section('content')
    <ol class='breadcrumb'>
        <li><a href='/'>Home</a></li>
        <li><a href='/lesson'>Lessons</a></li>
        <li class='active'>Create</li>
    </ol>

    <h2>Create a New Lesson</h2>

    <hr/>

    <form class="form"
        role="form"
        method="POST"
        action="{{ url('/lesson') }}">

        {{ csrf_field() }}

        <!-- lesson name Form Input -->

        <div class="form-group{{ $errors->has('name') ? ' has-error' : '' }}>

            <label class="control-label">
                Lesson Name
            </label>

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

                @if ($errors->has('name'))
```

```
<span class="help-block">  
    <strong>{{ $errors->first('name') }}</strong>  
</span>  
  
@endif  
</div>  
  
<div class="form-group">  
    <button type="submit"  
           class="btn btn-primary btn-lg">  
        Create  
    </button>  
</div>  
</form>  
  
@endsection
```

lesson/edit.blade.php

Gist:

[edit.blade.php](#)

From book:

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

@section('title')
    <title>Edit Lesson</title>

@endsection

@section('content')

    <ol class='breadcrumb'>
        <li><a href='/'>Home</a></li>
        <li><a href='/lesson'>Lessons</a></li>
        <li><a href='/lesson/{{ $lesson->id }}'>
            {{$lesson->name}}</a></li>
        <li class='active'>Edit</li>
    </ol>

    <h1>Edit Lesson</h1>

    <hr/>

    <form class="form"
        role="form"
        method="POST"
        action="{{ url('/lesson/'. $lesson->id) }}">

        {{ method_field('PATCH') }}

        {{ csrf_field() }}

        <!-- lesson name Form Input -->

        <div class="form-group{{ $errors->has('name') ? ' has-error' : '' }}>

            <label class="control-label">
                Lesson Name
            </label>

            <input type="text"
```

```
        class="form-control"
        name="name"
        value="{{ $lesson->name }}"

@if ($errors->has('name'))

<span class="help-block">

<strong>{{ $errors->first('name') }}</strong>

</span>

@endif

</div>

<div class="form-group">

<button type="submit"
        class="btn btn-primary btn-lg">

    Edit

</button>

</div>

</form>

@endsection
```

lesson/index.blade.php

Gist:

[index.blade.php](#)

From book:

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

@section('title')
    <title>Lessons</title>

@endsection

@section('content')
    <ol class='breadcrumb'>
        <li><a href='/'>Home</a></li>
        <li class='active'>Lessons</li>
    </ol>

    <h2>Lessons</h2>

    <hr/>

    <lesson-grid></lesson-grid>

    <div> <a href="/lesson/create">
        <button type="button"
            class="btn btn-lg btn-primary">
            Create New
        </button>
    </a>
    </div>

    @endsection
```

lesson/show.blade.php

Gist:

[show.blade.php](#)

From book:

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

@section('title')
    <title>{{ $lesson->name }} Lesson</title>

@endsection

@section('content')
    <ol class='breadcrumb'>
        <li><a href='/'>Home</a></li>
        <li><a href='/lesson'>lesson</a></li>
        <li><a href='/lesson/{{ $lesson->id }}'>
            {{ $lesson->name }}</a></li>
    </ol>

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

    <hr/>

    <div class="panel panel-default">
        <!-- Table -->
        <table class="table table-striped">

            <thead>
                <tr>
                    <th>Id</th>
                    <th>Name</th>
                    <th>Date Created</th>
                    <th>Edit</th>
                    <th>Delete</th>
                </tr>
            </thead>
```

```
<tbody>

<tr>
  <td>{{ $lesson->id }} </td>

  <td> <a href="/lesson/{{ $lesson->id }}/edit">
    {{ $lesson->name }}
  </a>

  </td>

  <td>
    {{ $lesson->created_at }}
  </td>

  <td> <a href="/lesson/{{ $lesson->id }}/edit">
    <button type="button"
            class="btn btn-default">
      Edit
    </button>
  </a>

  </td>

  <td>
    <div class="form-group">

      <form class="form"
            role="form"
            method="POST"
            action="{{ url('/lesson/'. $lesson->id) }}>

        <input type="hidden"
              name="_method"
              value="delete">
    
```

```
    {{ csrf_field() }}
```

```
    <input class="btn btn-danger"
          Onclick="return ConfirmDelete();"
          type="submit"
          value="Delete">
```

```
  </form>
```

```
  </div>
```

```
</td>
```

```
</tr>
```

```
</tbody>
```

```
</table>
```

```
</div>
```

```
@endsection
```

```
@section('scripts')
```

```
  <script>
```

```
    function ConfirmDelete()
    {
        var x = confirm("Are you sure you want to delete?");
        return x;
    }
  </script>
```

```
@endsection
```

That's it for our views, let's move on to our component:

LessonGrid.vue

Gist:

[LessonGrid.vue](#)

From book:

```
<template>

<div class="row">
  <div class="col-lg-12">
    <form id="search">
      Search
      <input name="query"
             v-model="query"
             @keyup="search(query)">
    </form>
    <div class="pull-right">
      {{ total }} Total Results
    </div>
  <section class="panel">
    <div class="panel-body">
      <table class="table table-bordered table-striped">
        <thead>
          <tr>
            <th v-for="key in gridColumns"
                @click="sortBy(key)"
                v-bind:class="{active: sortKey == key}">
```

```
    {{ key }}
```

```
    <span class="arrow"
          v-bind:class="sortOrder > 0 ? 'asc' : 'dsc'">
    </span>
```

```
</th>
```

```
<th>
```

```
  Actions
```

```
</th>
```

```
</tr>
```

```
</thead>
```

```
<tbody>
```

```
<tr v-for="row in gridData">
```

```
  <td>
```

```
    {{ row.Id }}
```

```
  </td>
```

```
  <td>
```

```
    <a v-bind:href="/lesson/" +row.Id+'-'+row.Slug ">
```

```
      {{ row.Name }}
```

```
    </a>
```

```
  </td>
```

```
  <td>
```

```
    {{ row.Created }}
```

```
  </td>
```

```
<td>

  <a v-bind:href="'/lesson/' +row.Id+'/edit'">

    <button type="button"
      class="btn btn-default">

      Edit

    </button>

  </a>

</td>

</tr>

</tbody>

</table>

</div>

<div class="pull-right">

  page {{ current_page }} of {{ last_page }} pages

</div>

</section>

<div class="row">

  <div class="pull-right for-page-button">

    <button @click="getData(go_to_page)"
      class="btn btn-default">

      Go To Page:

    </button>

  </div>
```

```
<input v-model="go_to_page"
       class="number-input"></input>

</div>

<ul class="pagination pull-right">

  <li>
    <a @click.prevent="getData(first_page_url)">
      first
    </a>
  </li>

  <li v-if="checkUrlNotNull(prev_page_url)">
    <a @click.prevent="getData(prev_page_url)">
      prev
    </a>
  </li>

  <li v-for="page in pages"
      v-if="page > current_page - 2 && page < current_page+2"
      v-bind:class="{'active': checkPage(page)}">
    <a @click.prevent="getData(page)">
      {{ page }}
    </a>
  </li>

  <li v-if="checkUrlNotNull(next_page_url)">
    <a @click.prevent="getData(next_page_url)">
      next
    </a>
  </li>
</ul>
```

```
</a>

</li>
<li><a @click.prevent="getData(last_page_url)">
    last
</a>
</li>
</ul>
</div>
</div>
</div>

</template>

<script>
export default {
    mounted: function () {
        this.loadData();
    },
    data: function () {
        return {
            query: '',
            gridColumns: ['Id', 'Name', 'Created'],
            gridData: [],
            total: null,
            next_page_url: null,
            prev_page_url: null,
            last_page: null,
            current_page: null,
        }
    }
}
```

```
    pages: [],
    first_page_url: null,
    last_page_url: null,
    go_to_page: null,
    sortOrder: 1,
    sortKey: ''

}

methods: {

  sortBy: function (key){

    this.sortKey = key;

    this.sortOrder = (this.sortOrder == 1) ? -1 : 1;

    this.getData(1);

  },

  search: function(query){

    this.getData(query);

  },

  loadData: function (){

    $.getJSON('api/lesson-data', function (data) {

      this.gridData = data.data;
      this.total = data.total;
      this.last_page = data.last_page;
      this.next_page_url = data.next_page_url;
      this.prev_page_url = data.prev_page_url;
      this.current_page = data.current_page;
      this.first_page_url = 'api/lesson-data?page=1';
      this.last_page_url = 'api/lesson-data?page=' +
      this.last_page;
      this.setPageNumbers();

    });

  }

};
```

```
        } .bind(this));
    },
    setPageNumbers: function(){
        for (var i = 1; i <= this.last_page; i++) {
            this.pages.push(i);
        }
    },
    getData: function (request){
        let getPage;
        switch (request){
            case this.prev_page_url :
                getPage = this.prev_page_url +
                    '&column=' + this.sortKey +
                    '&direction=' + this.sortOrder;
                break;
            case this.next_page_url :
                getPage = this.next_page_url +
                    '&column=' + this.sortKey +
                    '&direction=' + this.sortOrder;
                break;
            case this.first_page_url :
                getPage = this.first_page_url +
                    '&column=' + this.sortKey +
                    '&direction=' + this.sortOrder;
                break;
            case this.last_page_url :
```

```
getPage = this.last_page_url +
    '&column=' + this.sortKey +
    '&direction=' + this.sortOrder;

break;
```

case this.query :

```
getPage = 'api/lesson-data?' +
    'keyword=' + this.query +
    '&column=' + this.sortKey +
    '&direction=' + this.sortOrder;

break;
```

case this.go_to_page :

```
if( this.go_to_page != '' && this.pageInRange()) {

    getPage = 'api/lesson-data?' +
        'page=' + this.go_to_page +
        '&column=' + this.sortKey +
        '&direction=' + this.sortOrder +
        '&keyword=' + this.query;

    this.clearPageNumberInputBox();

} else {

    alert('Please enter a valid page number');

}
```

break;

default :

```
getPage = 'api/lesson-data?' +
    'page=' + request +
    '&column=' + this.sortKey +
    '&direction=' + this.sortOrder +
    '&keyword=' + this.query;

break;
```

```
        }

        if (this.query == '' && getPage != null){

            $.getJSON(getPage, function (data) {

                this.gridData = data.data;
                this.total = data.total;
                this.last_page = data.last_page;
                this.next_page_url = data.next_page_url;
                this.prev_page_url = data.prev_page_url;
                this.current_page = data.current_page;

            }).bind(this));

        } else {

            if (getPage != null){

                $.getJSON(getPage, function (data) {

                    this.gridData = data.data;
                    this.total = data.total;
                    this.last_page = data.last_page;
                    this.next_page_url = (data.next_page_url == null) ?
                        null : data.next_page_url + '&keyword=' +this.query;
                    this.prev_page_url = (data.prev_page_url == null) ?
                        null : data.prev_page_url + '&keyword=' +this.query;
                    this.first_page_url =
                        'api/lesson-data?page=1&keyword=' +this.query;
                    this.last_page_url = 'api/lesson-data?page=' +
                        this.last_page + '&keyword=' +this.query;
                    this.current_page = data.current_page;
                    this.resetPageNumbers();

                }).bind(this));

            }

        },
    },
```

```
checkPage: function(page){  
  return page == this.current_page;  
},  
  
resetPageNumbers: function(){  
  this.pages = [ ];  
  for (var i = 1; i <= this.last_page; i++) {  
    this.pages.push(i);  
  }  
},  
  
checkUrlNotNull: function(url){  
  return url != null;  
},  
  
clearPageNumberInputBox: function(){  
  return this.go_to_page = '';  
},  
  
pageInRange: function(){  
  return this.go_to_page <= parseInt(this.last_page);  
}  
}  
}  
  
</script>
```

Next will add the call to the component in the components file:

components.js

```
Vue.component('lesson-grid',
require('./components/LessonGrid.vue'));
```

ApiController

Let's add the following method to the ApiController:

Gist:

[ApiController.php](#)

From book:

```
public function lessonData(Request $request)
{
    return GridQuery::sendData($request, new LessonQuery);
}
```

Don't forget the use statement at the top of the file:

```
use App\Queries\GridQueries\LessonQuery;
```

Api Route for Lesson Model

In web.php, let's create the following route:

Gist:

[web.php](#)

From book:

```
Route::get('api/lesson-data', 'ApiController@lessonData');
```

LessonQuery

In app/Queries/GridQueries, let's create LessonQuery.php, with the following contents.

Gist:

[LessonQuery.php](#)

From book:

```
<?php

namespace App\Queries\GridQueries;

use DB;
use App\Queries\GridQueries\Contracts\DataQuery;

class LessonQuery implements DataQuery
{
    public function data($column, $direction)
    {
        $rows = DB::table('lessons')
            ->select('id as Id',
                      'name as Name',
                      'slug as Slug',
                      DB::raw('DATE_FORMAT(created_at,"%m-%d-%Y") as Created'))
    }
}
```

```
        ->orderBy($column, $direction)
        ->paginate(10);

    return $rows;

}

public function filteredData($column, $direction, $keyword)
{
    $rows = DB::table('lessons')
        ->select('id as Id',
                  'name as Name',
                  'slug as Slug',
                  DB::raw('DATE_FORMAT(created_at,"%m-%d-%Y") as Created'))
        ->where('name', 'like', '%' . $keyword . '%')
        ->orderBy($column, $direction)
        ->paginate(10);

    return $rows;
}

}
```

Let's run the following from the command line:

```
npm run dev
```

And our Lesson model is setup. If you visit:

```
sample-project.com/lesson
```

You should see an empty grid. We didn't make a factory because there was no point in populating it with a bunch of latin words. Plus the category and subcategory association is not yet enforced, so we can end up with records that don't make sense.

We could overcome this by writing code for our lesson factory that would do what we want, but that is not the point of this chapter.

If you notice, on our lesson/create.blade.php, we don't have anyway to send along the category and subcategory, and both are ignored by the controller.

So let's start there and we will slowly dial this towards what we want. Let's add the following to our lesson/create.blade.php: Gist:

create.blade.php

From book:

```
<!-- category select -->

<div class="form-group{{ $errors->has('category_id') ? ' has-error' : '' }}>

    <label class="control-label">
        Category
    </label>

    <select class="form-control"
            id="category_id"
            name="category_id">

        <option value="">Please Choose One</option>

        @foreach($categories as $category)

            <option value="{{ $category->id }}>
                {{ $category->name }}
            </option>

        @endforeach

    </select>
```

```
@if ($errors->has('category_id'))  
  
    <span class="help-block">  
  
        <strong>  
  
            {{ $errors->first('category_id') }}  
  
        </strong>  
  
    </span>  
  
@endif  
  
</div>  
  
<!-- subcategory select -->  
  
<div class="form-group{{ $errors->has('subcategory_id') ? ' has-error' : '' }}"  
  
    " >  
  
    <label class="control-label">  
  
        Subcategory  
  
    </label>  
  
<select class="form-control"  
       id="subcategory_id"  
       name="subcategory_id">  
  
    <option value="">Please Choose One</option>  
  
    @foreach($subcategories as $subcategory)  
  
        <option value="{{ $subcategory->id }}>  
            {{ $subcategory->name }}  
    </option>  
    @endforeach  
    </select>
```

```

        </option>

    @endforeach

</select>

@if ($errors->has('subcategory_id'))


{{ $errors->first('subcategory_id') }}


@endif

</div>

```

This still isn't going to work, however, because we need to send along our \$categories and \$subcategories objects.

So let's modify the create method of the LessonController to the following:

Gist:

[LessonController.php](#)

From book:

```

public function create()
{
    $categories = Category::all();
    $subcategories = Subcategory::all();

    return view('lesson.create',
        compact('categories', 'subcategories'));
}

```

That will get us the form with two dropdown selects, one for category, one for subcategory:

The screenshot shows a 'Create a New Lesson' form. At the top, there's a navigation bar with links for Home, About, Users, Content, and a user profile for 'gumby'. Below the navigation is a breadcrumb trail: Home / Lessons / Create. The main form area has three input fields: 'Lesson Name' (empty), 'Category' (a dropdown menu showing 'Please Choose One'), and 'Subcategory' (a dropdown menu showing 'Please Choose One'). A blue 'Create' button is located at the bottom left of the form. At the very bottom, a small footer note reads: © 2017 Sample Project All rights Reserved.

The good news is that we have both dropdowns with values. The bad news is that they have no knowledge of each other, so you can select a category like php, than select a subcategory like hoisting or ES6, or any subcategory that belongs to javascript and you can see the problem.

Just for diligence, we should dd(\$request) in the LessonController's store method, just to make sure the values are being passed along.

So the first part of the method looks like this:

```
public function store(Request $request)
{
    dd($request);
```

Next try to create a lesson record for with the name of PHP 101. After you click on the down arrows for +request: ParameterBag and #parameters:, you should get output that looks something like:

The screenshot shows a web application interface for creating a new lesson. At the top, there's a navigation bar with links for Home, About, Users, Content, and a user profile for 'gumby'. Below the navigation is a breadcrumb trail: Home / Lessons / Create. The main content area is titled 'Create a New Lesson'. It contains three input fields: 'Lesson Name' (empty), 'Category' (dropdown menu showing 'Please Choose One'), and 'Subcategory' (dropdown menu showing 'Please Choose One'). A blue 'Create' button is positioned below these fields. In the bottom right corner of the page, there's a small copyright notice: '© 2017 Sample Project All rights Reserved.'

We can see the values we chose are being passed along.

What we want now is to make validation that will not accept this request, unless the subcategory_id does indeed belong to the category_id.

So far in our store method on LessonController.php, we are only validating a single input for 'name.' So obviously this is not going to work.

Custom Validation

Let's create a form request object to handle the validation, so we can work in some custom rules.

Lesson Create Request

Let's use artisan to make the request file:

```
php artisan make:request LessonCreateRequest
```

Now let's change the LessonCreateRequest stub to the following:

Gist:

[LessonCreateRequest](#)

From book:

```
<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;
use Illuminate\Http\Request;
use App\Category;
use App\Subcategory;
use Illuminate\Validation\Factory as ValidationFactory;

class LessonCreateRequest extends FormRequest
{

    public function __construct(ValidationFactory $validationFactory)
    {

        $validationFactory->extend(
            'belongsToCategory',
            function ($attribute, $value, $parameters) {
                $category_id = request()->input('category_id');

                $verified = Subcategory::where('id', $value)
                    ->where('category_id', $category_id)
                    ->exists();

                return 'belongsToCategory' == $verified;
            },
            'Sorry, subcategory does not belong to category!'
        );
    }

    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
}
```

```

public function authorize()
{
    return true;
}

/**
 * Get the validation rules that apply to the request.
 *
 * @return array
 */

public function rules()
{
    $count = Category::count();

    return [
        'name' => 'required|unique:lessons|string|max:30',
        'category_id' =>
            "required|numeric|digits_between:1,$count",
        'subcategory_id' => 'required|belongsToManyCategory'
    ];
}

}

```

Ok, you can see we brought in some additional use statements:

```

use App\Category;
use App\Subcategory;
use Illuminate\Validation\Factory as ValidationFactory;

```

Next we, added a constructor that pulls in an instance of ValidationFactory:

```

public function __construct(ValidationFactory $validationFactory)
{
    $validationFactory->extend(
        'belongsToCategory',
        function ($attribute, $value, $parameters) {
            $category_id = request()->input('category_id');

            $verified = Subcategory::where('id', $value)
                ->where('category_id', $category_id)
                ->exists();

            return 'belongsToCategory' == $verified;
        },
        'Sorry, subcategory does not belong to category!'
    );
}

```

We use the extend method of the ValidationFactory instance to create the new rule, which we are calling 'belongsToCategory.' The second argument of the extend method is a closure, taking in \$attribute, \$value, and \$parameters, where we do our logic to test if the subcategory belongs to the category.

We test the result of a query of Subcategory against the request() value to see if it exists. This will return true or false for the \$verified variable. Then we use the return statement like so:

```
return 'belongsToCategory' == $verified;
```

We are also able to send along an error message as the third parameter:

'Sorry, subcategory does not belong to category!'

Now that we have this setup, we can use our belongsToCategory rule in the rules array:

```
public function rules()
{
    $count = Category::count();

    return [
        'name' => 'required|unique:lessons|string|max:30',
        'category_id' =>
            "required|numeric|digits_between:1,$count",
        'subcategory_id' => 'required|belongsToCategory'

    ];
}
```

Also note we are getting the count of all the category records, so we can use it as a parameter in our digits_-between validation rule:

'category_id' => "required|numeric|digits_between:1,\$count",

Note the use of double quotes, so we can interpolate the variable.

Because we left the dd(\$request) in place on the controller, you can play around with this. When you choose combinations of category and subcategory, try a few that don't belong together and you will see that it fails validation.

So all that should be working, but this is not the best solution for our current scenario. We are going to need the same validation rule on our LessonController update method, so it doesn't make sense to have to duplicate code.

Validator Service Provider

We can avoid code duplication by chopping out the constructor out of our LessonCreateRequest class.

Instead of a method in the constructor, we are going to create a separate service provider, so this rule can be accessible everywhere.

Let's start by making the new provider with artisan:

```
php artisan make:provider ValidatorServiceProvider
```

Now let's change the contents of the stub to:

Gist:

[ValidatorServiceProvider.php](#)

From book:

```
<?php
```

```
namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use App\Subcategory;

class ValidatorServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap the application services.
     *
     * @return void
     */

    public function boot()
    {
        $this->app['validator']->extend('belongsToCategory',
            function ($attribute, $value, $parameters) {
                $category_id = request()->input('category_id');
```

```
$verified = Subcategory::where('id', $value)
    ->where('category_id', $category_id)
    ->exists();

return 'belongsToCategory' == $verified;

}

'Sorry, subcategory does not belong to category!');

}

/**
 * Register the application services.
 *
 * @return void
 */

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

This time we are using a slightly less verbose way of calling the class by using `$this->app['validator']`, which, because an alias for ‘validator’ is registered in our config/app.php aliases array, points to a facade that will instantiate the class.

Other than that difference, it is exactly the same, and all we need to do now is add the ValidatorServiceProvider to our config/app.php providers array.

```
/*
 * Application Service Providers...
 */
App\Providers\AppServiceProvider::class,
App\Providers\AuthServiceProvider::class,
App\Providers\BroadcastServiceProvider::class,
App\Providers\EventServiceProvider::class,
App\Providers\RouteServiceProvider::class,
App\Providers\ValidatorServiceProvider::class,
//App\Providers\RocketShipServiceProvider::class,
```

And your rule should work now.

Multiple Custom Validation Rules

To add more validation rules to the provider, you would simply add another method call in the boot method. Since we want to make sure the category_id is within the range of actual category ids, we may as well make a rule for this.

Let's start by changing the rules method on our LessonCreateRequest object:

```
public function rules()
{
    return [
        'name' => 'required|unique:lessons|string|max:30',
        'category_id' => 'required|numeric|isValidCategory',
        'subcategory_id' => 'required|belongsToManyCategory'
    ];
}
```

Next, modify your boot method on ValidatorServiceProvider to the following:

Gist:

[ValidatorServiceProvider.php](#)

From book:

```
public function boot()
{
    $this->app['validator']->extend('isValidCategory',
        function ($attribute, $value, $parameters){
            $max = Category::count();
            $verified = (1 <= $value) && ($value <= $max) ? true : false;
            return 'isValidCategory' == $verified;
        },
        'Sorry, category is not a valid category!');

    $this->app['validator']->extend('belongsToCategory',
        function ($attribute, $value, $parameters){
            $category_id = request()->input('category_id');
            $verified = Subcategory::where('id', $value)
                ->where('category_id', $category_id)
                ->exists();
            return 'belongsToCategory' == $verified;
        },
        'Sorry, subcategory does not belong to category!');
}
```

You can see we just added the extend method call for ‘isValidCategory’.

Inside that method, we check to see if the given category_id falls between 1 and the count number, which represents the highest number possible:

```
$max = Category::count();  
  
$verified = (1 <= $value) && ($value <= $max) ? true : false;  
  
return 'isValidCategory' == $verified;
```

And that's it, we now have 2 custom validation rules.

Now let's finish up our store method, so we can actually record a couple of records. Change the store method on LessonController.php to the following:

Gist:

[LessonController.php](#)

From book:

```
public function store(LessonCreateRequest $request)  
{  
  
    $slug = str_slug($request->name, "-");  
  
    $widget = Lesson::create([  
  
        'name' => $request->name,  
        'slug' => $slug,  
        'category_id' => $request->category_id,  
        'subcategory_id' => $request->subcategory_id  
  
    ]);
```

```

$widget->save();

alert()->success('Congrats!', 'You made a Lesson');

return Redirect::route('lesson.index');

}

```

So now you should be able to create a record.

Left Join

Ok, so things are moving along, but you can see that our index page does not display the category or subcategory, so we need to do a couple of revisions to get that to work.

We need to add a couple of left joins to our queries in our LessonQuery class.

LessonQuery.php Revised

Let's replace LessonQuery.php with the following, and definitely use the gist for code formatting:

Gist:

[LessonQuery.php](#)

From book:

```

<?php

namespace App\Queries\GridQueries;

use DB;
use App\Queries\GridQueries\Contracts\DataQuery;

class LessonQuery implements DataQuery
{
    public function data($column, $direction)
    {
        $rows = DB::table('lessons')
            ->select('lessons.id as Id',

```

```

        'lessons.name as Name',
        'categories.name as Category',
        'subcategories.name as Subcategory',
        'lessons.slug as Slug',
    DB::raw('DATE_FORMAT(lessons.created_at,"%m-%d-%Y")'
           as Created))
    ->leftJoin('categories', 'category_id', '=', 'categories.id')
    ->leftJoin('subcategories', 'subcategory_id', '=', 'subcategories.id')
    ->orderBy($column, $direction)
    ->paginate(10);

    return $rows;
}

public function filteredData($column, $direction, $keyword)
{
    $rows = DB::table('lessons')
        ->select('lessons.id as Id',
                  'lessons.name as Name',
                  'categories.name as Category',
                  'subcategories.name as Subcategory',
                  'lessons.slug as Slug',
        DB::raw('DATE_FORMAT(lessons.created_at,"%m-%d-%Y")'
               as Created))
        ->leftJoin('categories', 'category_id', '=', 'categories.id')
        ->leftJoin('subcategories', 'subcategory_id', '=', 'subcategories.id')
        ->where('name', 'like', '%' . $keyword . '%')
        ->orderBy($column, $direction)
        ->paginate(10);

    return $rows;
}
}

```

So you can see in this case, since we are using more than one table, we have to disambiguate the select statement by prefixing the table name to the select columns. That's why we see lessons.name as Name.

Ok, let's move on to the left joins. The definition straight from W3 schools:

The LEFT JOIN keyword returns all rows from the left table (table1), with the matching rows in the right table (table2). The result is NULL in the right side when there is no match.

Obviously, in our case, we are using 2 left joins, one for category and one for subcategory:

```
->leftJoin('categories', 'category_id', '=', 'categories.id')
->leftJoin('subcategories', 'subcategory_id', '=', 'subcategories.id')
```

You can see that category_id on the lessons table is being mapped to the id on the categories table. Subcategories are done in a similar fashion. So that's how we stitch the tables together using the query builder.

Just a note for troubleshooting. If your grid displays with no data, go to the following url:

sample-project.com/api/lesson-data

Because we have not constrained the controller method to ajax-only calls, like we did with our Chat controller, we can test it this way. If there is a problem with your query, you will see a helpful error message. Otherwise, in the console, you will only get a 500 internal server error, which is not that helpful.

LessonGrid.vue Revised

Next we need to make a couple of changes to our LessonGrid.vue file.

Let's start with changing the gridColumn array inside of our data object:

```
gridColumns: ['Id', 'Name', 'Category', 'Subcategory', 'Created'],
```

Then we need to add 2 <td> sections in the template, under the row.name:

```
<td>
  {{ row.Category }}
</td>
<td>
  {{ row.Subcategory }}
</td>
```

Here is the gist of the [entire file](#) if you need it for reference.

So now, assuming you have a record created, you should see something like the following on the index page:

The screenshot shows a Laravel application interface. At the top, there is a dark header bar with the text "Sample Project" on the left and navigation links "Home", "About", "Users", "Content", "gumby", and a user profile icon on the right. Below the header, the URL "Home / Lessons" is displayed. The main content area has a title "Lessons". A search bar labeled "Search" is followed by a table with 1 total result. The table has columns: Id, Name, Category, Subcategory, Created, and Actions. The single row shows Id 1, Name "PHP 101", Category "PHP", Subcategory "public method", Created "02-12-2017", and an "Edit" button in the Actions column. At the bottom, there are pagination links "first", "1", "last", and a "Go To Page:" input field. A blue "Create New" button is located at the bottom left of the main content area.

| Id | Name | Category | Subcategory | Created | Actions |
|----|---------|----------|---------------|------------|-----------------------|
| 1 | PHP 101 | PHP | public method | 02-12-2017 | <button>Edit</button> |

Lesson Show Revised

We also need to fix the show page. Let's add to `lesson/show.blade.php` table heading with the following:

```
<th>Id</th>
<th>Name</th>
<th>Category</th>
<th>Subcategory</th>
<th>Date Created</th>
<th>Edit</th>
<th>Delete</th>
```

And to the body:

```
<td>{{ $subcategory->name }}</td>
<td>{{ $lesson->created_at }}</td>
```

So now your create, index, and show views should be working for /lesson. We did not work on edit because we still have the problem of the dropdown. While we're validating that a subcategory must belong to a user, we still allow the user to make a mistake, if they choose the wrong subcategory.

It would be far better for us to limit the subcategory to the correct choices that they have in the first place.

Dependent Dropdown

We're going to create a Vue.js component to handle our dropdowns. We're going to replace the entire dropdown sections we currently have in lesson/create.blade.php with the following:

```
<lesson-create-category
:categories="{{ json_encode($categories) }}"
:subcategories="{{ json_encode($subcategories) }}"
>
</lesson-create-category>
```

Here is the [entire create.blade.php file](#) for reference.

One thing we are doing differently here is that we are binding in the objects that we pass from the controller into blade into the component, using json_encode to make sure the object is formatted for javascript.

The other way we could do that is to make an ajax call from within our component, and for large datasets, that would be the appropriate way to go, since the way we are doing it here, we are going to get all our data at once, not in paginated chunks.

But for small data sets, like a category list and subcategories, this is fine, and requires less code. It's a fast simple way to bring in the data.

LessonCreateCategory.vue

Let's create the following component in resources/assets/js/LessonCreateCategory.vue:

Gist:

[LessonCreateCategory.vue](#)

From book:

```
<template>

<div>

    <!-- Category Select Form -->

    <div class="form-group">

        <label for="category_id">

            Category Name:

        </label>

        <select @change="onChange($event.target.value)"
               class="form-control"
               name="category_id">

            <option value="">Please Choose One</option>

            <option v-for="category in categories"
                   v-bind:value="category.id">


```

```
>{{ category.name }}</option>

</select>

</div>

<!-- Subcategory Select Form -->

<div class="form-group" v-if="category > 0">

<label for="subcategory_id">Subcategory Name:</label>

<select v-model="defaultOption"
        class="form-control"
        name="subcategory_id">

<option value="0">Please Choose One</option>

<option v-for="option in options"
        v-bind:value="option.id">
    {{ option.name }}</option>

</select>

</div>

</div>

</template>

<script>

export default {

  props: ['subcategories', 'categories'],

  data () {

    return {

      category : '',

```

```
        options: [],
        defaultOption: 0
    },
},
methods: {
    filterSubcategories () {
        this.options = this.subcategories
            .filter(subcategories =>
                subcategories.category_id == this.category);
    },
    onChange (value) {
        this.category = value;
        this.defaultOption = 0;

        this.filterSubcategories();
    }
}
</script>
```

Before we step through it, let's add the call to resources/assets/js/components.js

```
Vue.component('lesson-create-category',  
require('./components/LessonCreateCategory.vue'));
```

Let's run the following from the command line to get it working:

```
npm run dev
```

Ok, so let's look at the component script first. You can see we are bringing the properties from the tag on create.blade.php:

```
export default {  
  
  props: ['subcategories', 'categories'],
```

That takes care of consuming in the data, which is pretty simple.

Next we have our data function:

```
data () {  
  
  return {  
  
    category: '',  
  
    options: [],  
  
    defaultOption: 0  
  
  },
```

The category attribute will represent the selected category. The options array represents the filtered subcategories, and defaultOption represents the “Please choose one” option on the subcategory options. We are setting that to 0 as an initial state.

Next we have our methods:

```
methods: {  
  
    filterSubcategories () {  
  
        this.options = this.subcategories  
            .filter(subcategories =>  
                subcategories.category_id == this.category);  
  
    },  
  
    onChange (value) {  
  
        this.category = value;  
  
        this.defaultOption = 0;  
  
        this.filterSubcategories();  
  
    }  
}
```

So we have a method to filter the subcategories, limiting them to only those that match the filter. We use the native javascript filter method for this:

```
.filter(subcategories => subcategories.category_id == this.category);
```

We learned about that function when we built our chat room.

Next we have our onChange method, which is listening for changes in the category select:

```
onChange (value) {  
  
  this.category = value;  
  
  this.defaultOption = 0;  
  
  this.filterSubcategories();  
  
}
```

You can see we are handing it the value of the selected option, which is then set to the property category. We make sure our defaultOption is set to 0 because before I added that, we had unexpected behavior when the category was selected. For some reason the subcategory would get selected too, which is not what we wanted, so we force it to 0 here. Then we call our filterSubcategories method, which will filter the subcategories and load them into our options array.

Now when we look at the template, this is going to make a lot of sense. Let's look at category first:

```
<!-- Category Select Form -->  
  
<div class="form-group">  
  
  <label for="category_id">Category Name:</label>  
  
  <select @change="onChange($event.target.value)"  
         class="form-control"  
         name="category_id">
```

```
<option value="">Please Choose One</option>

<option v-for="category in categories"
        v-bind:value="category.id"
        >{{ category.name }}</option>

</select>

</div>
```

Only two things of note here. First is the event:

```
<select @change="onChange($event.target.value)"
```

We are using @change, which calls our onChange method, handing in the \$event.target.value, which is the value that is selected.

To get our list of categories, which comes directly from the properties we brought in, we simply use a v-for directive:

```
<option v-for="category in categories"
        v-bind:value="category.id"
        >{{ category.name }}</option>
```

And that gets us our list of categories to select.

Ok, on to our subcategories:

```
<!-- Subcategory Select Form -->

<div class="form-group" v-if="category > 0">

  <label for="subcategory_id">Subcategory Name:</label>

  <select v-model="defaultOption"
    class="form-control"
    name="subcategory_id">

    <option value="0">Please Choose One</option>

    <option v-for="option in options"
      v-bind:value="option.id">
      {{ option.name }}</option>

  </select>

</div>
```

The first thing to note is that we are using a v-if directive to determine whether or not to show the div. We only get to see it if category is greater than 0, which only happens if a category has been selected. That's a lot of power in very little code.

Next we bind the v-model="defaultOption" to the select. This gives us control over the initial state, which we are setting to 0. You can see the option with a value of 0 is "Please choose one."

Then to get our list of options, we use v-for to iterate over the options, which we have set from the filtered subcategories.

All this should be working now. But we lost some functionality. We are not getting our errors back. If you try to submit without choosing category or subcategory, it returns you to page without feedback.

There are two approaches you can wire up for this. One is more complicated, and that would be to handle the validation for the dropdowns in javascript, within the component or by creating a special javascript errors class to handle the errors. I'm not going to demonstrate that, since that would take us too far off into javascript, and this is a book about Laravel.

So I'm going for option # 2, which is to include the errors.list partial in lesson/create.blade.php and let that provide the feedback. So right below the last closing </form> tag in lesson/create.blade.php, place the following:

```
@include('errors.list')
```

And that's it!

lesson/edit.blade.php

Ok, one last view for this chapter, the edit view. We are going to move quickly on this one. Let's make lesson/edit.blade.php as follows:

Gist:

[edit.blade.php](#)

From book:

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

@section('title')
    <title>Edit Lesson</title>
@endsection

@section('content')

    <ol class='breadcrumb'>
        <li><a href='/'>Home</a></li>
        <li><a href='/lesson'>Lessons</a></li>
        <li><a href='/lesson/{{ $lesson->id }}'>
            {{$lesson->name}}</a></li>
        <li class='active'>Edit</li>
    </ol>

    <h1>Edit Lesson</h1>

    <hr/>

    <form class="form"
          role="form"
          method="POST"
```

```
action="{{ url('/lesson/'. $lesson->id) }}"
{{ method_field('PATCH') }}
{{ csrf_field() }}

<!-- lesson name Form Input -->

<div class="form-group{{ $errors->has('name') ?
    ' has-error' : '' }}>

    <label class="control-label">
        Lesson Name
    </label>

    <input type="text"
        class="form-control"
        name="name"
        value="{{ $lesson->name }}>

    @if ($errors->has('name'))
        <span class="help-block">
            <strong>
                {{ $errors->first('name') }}
            </strong>
        </span>
    @endif
</div>

<lesson-edit-category :category="{{ json_encode($category) }}"
    :categories="{{ json_encode($categories) }}"
    :subcategory="{{ json_encode($subcategory) }}"
    :subcategories="{{ json_encode($subcategories) }}>

</lesson-edit-category>
```

```
<div class="form-group">

    <button type="submit"
        class="btn btn-primary btn-lg">
        Edit
    </button>
</div>

</form>

@include('errors.list')

@endsection
```

Ok, we can see we're passing in the objects to the controller, but our edit and store methods still need work:

Gist:

[LessonController.php](#)

From book:

```
public function edit(Lesson $lesson)
{
    $category = Category::where('id', $lesson->category_id)->first();

    $subcategory = Subcategory::where('id', $lesson->subcategory_id)
        ->first();

    list($categories, $subcategories) = $this->getCategoryLists();

    return view('lesson.edit', compact('lesson',
        'category',
        'subcategory',
```

```
        'categories',
        'subcategories')));
}

/**
 * Update the specified resource in storage.
 *
 * @param \Illuminate\Http\Request $request
 * @param int $id
 * @return \Illuminate\Http\Response
 */

public function update(Request $request, Lesson $lesson)
{
    $this->validate($request, [
        'name' => 'required|string|max:40|unique:widgets,name,' .
            $lesson->id,
        'category_id' => 'required|numeric|isValidCategory',
        'subcategory_id' => 'required|belongsToCategory'
    ]);

    $slug = str_slug($request->name, "-");

    $lesson->update([
        'name' => $request->name,
        'slug' => $slug,
        'category_id' => $request->category_id,
        'subcategory_id' => $request->subcategory_id
    ]);

    alert()->success('Congrats!', 'You updated a lesson');

    return Redirect::route('lesson.show', [
        'lesson' => $lesson,
        'slug' => $slug
    ]);
}
```

I provided the full controller in the gist, check that if you have an issue. You can see we are doing the validation in the update method, using our custom validators, it all works beautifully. Other than there is nothing new except we are using a method in the edit method to list \$categories and \$subcategories:

```
private function getCategoryLists()
{
    $categories = Category::all();
    $subcategories = Subcategory::all();
    return [$categories, $subcategories];
}
```

Since we use these same objects for the create method, we made a method that will return the values we need, so we can use them in both create and edit, and avoid code duplication.

Just for some cleanup, I changed the constructor too:

```
public function __construct()
{
    $this->middleware('auth', ['except' => 'index']);
    $this->middleware('admin', ['except' => ['index', 'show']]);
}
```

LessonEditCategory.vue

Now we need our component. Let's create LessonEditCategory.vue with the following:

Gist:

[LessonEditCategory.vue](#)

From book:

```
<template>

<div>

<!-- Category Select Form -->

<div class="form-group">

  <label for="category_id">Category Name:</label>

  <select @change="onChange($event.target.value)"
          class="form-control"
          name="category_id">

    <option v-bind:value="category.id">
      {{ category.name }}
    </option>

    <option v-for="category in categories"
           v-bind:value="category.id">
      {{ category.name }}
    </option>

  </select>

</div>

<!-- Subcategory Select Form -->

<div class="form-group"
     v-if="category.id > 0">

  <label for="subcategory_id">
    Subcategory Name:
  </label>

  <select v-model="defaultOption">
```

```
        class="form-control"
        name="subcategory_id">

    <option v-bind:value="subcategory.id">
        {{ subcategory.name }}
    </option>

    <option value="0">Please Choose One</option>

    <option v-for="option in options"
            v-bind:value="option.id">
        {{ option.name }}
    </option>

</select>

</div>

</div>

</template>

<script>

export default {

    props: [ 'category',
        'subcategory',
        'subcategories',
        'categories' ],

    data () {

        return {
            newSelectedCategory : ''
        }
    }

}
```

```
options: [],

defaultOption: this.subcategory.id


}

methods: {

filterSubcategories () {

    this.options = this.subcategories
    .filter(subcategories =>
        subcategories.category_id == this.newSelectedCategory);

},
onChange (value) {

    this.newSelectedCategory = value;

    this.defaultOption = 0;

    this.filterSubcategories();

}

}

}

</script>
```

Ok, we made just a few tweaks to account for displaying the old selection from the db. That's why we have the additional properties coming in via the component.

```
props: ['category', 'subcategory', 'subcategories', 'categories'],
```

Because we are pulling in the old category and calling it category, which seems logical, I had to change the name of the selected value of category, which is only changed when the user decides to do so, to newSelectedCategory:

```
data () {  
  return {  
    newSelectedCategory: '',  
    options: [],  
    defaultOption: this.subcategory.id  
  }  
},
```

That's start out with an empty string because it might not be needed if the user is not changing the category. Then in the template, we make sure to include as the first option the category property, which has the old values for category:

```
<option v-bind:value="category.id">{{ category.name }}</option>
```

And the one for subcategory:

```
<option v-bind:value="subcategory.id">{{ subcategory.name }}</option>
```

Ok, let's remember to add the call to the component in components.js:

```
Vue.component('lesson-edit-category',  
  require('./components/LessonEditCategory.vue'));
```

Here is the entire components file, so you have it for reference:

[components.js](#)

Summary

It took a lot of pages to build up that example, but I think it was worth it. We learned more about custom validation, left joins in the query builder, and did more with Vue.js, learning how to take in objects passed by blade into a component, to make a dependent dropdown.

I hope this book has helped you with both Laravel and Vue.js. Please leave a rating or review at [GoodReads.com](#), I really appreciate it.

Thanks again for buying the book and supporting my work. See you soon.

Bill.