EASY E-COMMERCE

Using Laravel and Stripe

Selling Products and Subscriptions

easyecommercebook.com

Eric L. Barnes and W. Jason Gilmore

Easy E-Commerce Using Laravel and Stripe

Selling Products and Subscriptions

W. Jason Gilmore and Eric L. Barnes

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

This version was published on 2015-08-06

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.

Tweet This Book!

Please help W. Jason Gilmore and Eric L. Barnes by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought ”Easy E-Commerce Using Laravel and Stripe” from @leanpub!

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#

Also By These Authors

Books by W. Jason Gilmore

Easy Active Record for Rails Developers

Easy Laravel 5

Easy React

Books by Eric L. Barnes

The Artisan Files

I dedicate this book to my wife, Shannon. Thank you for all you do. - Eric

I dedicate this book to my grandparents for putting that Commodore 64 under the Christmas tree so many years ago. I miss you. - Jason

Contents

Introduction

The excitement associated with building an online store with the potential to attract a global customer base can be truly palpable. Emotions can run even higher when the project is associated with a small business or startup given the real possibility of a life-changing outcome should the store be successful.

However, these sorts of projects present a double-edged sword in that the sense of adventure is typically accompanied by a great deal of stress. Is the product catalog accessible and presenting the products in the best possible light? Is the credit card processor working flawlessly not only right now but at 2am when the development team is sleeping? Will the store owner receive timely updates regarding purchases and relevant sales trends? These are just a few questions that tend to bestow the gift of insomnia upon newcomers to the world of e-commerce development.

Fortunately, the Laravel community is blessed with an incredibly powerful framework which can effectively put many of these concerns (along with exhausted developers) to bed. Offering a well-organized application structure, sane forms processing features, PHPUnit integration, and even an interface for selling products and subscriptions through the popular Stripe[1] payment processing solution, you'll be able to create practical and maintainable e-commerce solutions under even the most demanding schedules.

In this book you'll learn how to use the Laravel framework and Stripe to create a fairly simple yet most certainly real-world online store. You'll learn by doing, following along with the creation of an e-commerce site for a fictitious lawn care company named We Dew Lawns, Inc. In addition to learning how the company website http://wedewlawns.com[2] was developed, we'll really try to inject a sense of realism into the example project by concluding each phase with an exit interview in which the surly company owner asks you key questions about the implementation, so you'd better be prepared!

Although we will be building a new Laravel site from the ground up, we do presume you are at least acquainted with Laravel 5 fundamentals. If your are new to Laravel or are in need of a refresher, consider purchasing a copy of Jason's bestselling book, *Easy Laravel 5*. Learn more about the book at http://easylaravelbook.com[3].

[1] https://stripe.com/

[2] http://wedewlawns.com

[3] http://easylaravelbook.com

About the Book Project

We Dew Lawns (http://wedewlawns.com[4]) is a fictional multi-generational family business who has historically promoted their services through traditional media, including the phone book, mailbox flyers, and local newspaper advertisements. Grandson Todd McDew has recently taken the reins, and although he's the first to admit being more comfortable wrenching on a lawn mower than typing on a keyboard, he's determined to bring the company into the 21st century.

After several meetings, Todd has concluded he would like to work with you on creating an official company website. However, in order to minimize risk and expense, he insists the project be completed in several phases. If after the completion of this first phase he declares satisfaction, you'll be hired on to complete the next phase.

Each chapter of this book guides you through a new project phase. At the conclusion of each chapter you'll be quizzed by Mr. McDew regarding various implementation decisions made throughout the phase, and if you can sufficiently respond to his questions you'll be invited to work on the next phase. Although we will provide example responses to his questions, we encourage you to think about these and other questions a potential client might ask you on a future project!

Below we'll introduce each phase by way of its corresponding chapter description.

Chapter 1. Creating the We Dew Lawns Website

In this opening phase you'll be tasked with the creation of a fairly simple Laravel-driven website that tells prospective customers more about the business, and offers a user-friendly contact form which sends inquiries to the company assistant, Patty Organo.

To suit this requirement, you'll work through the creation of a new Laravel 5 website complete with Less, Elixir, and Bootstrap integration. You'll also create a home and company bio pages, a site-wide layout, and a Mandrill[5]-backed contact form.

Chapter 2. Integrating User Accounts

Following a successful launch of the first phase, Mr. McDew would like to begin building a customer mailing list. To entice individuals into handing over their valuable e-mail addresses he would like to offer registered users access to downloadable coupons which they can apply to lawn care services. In this chapter you'll integrate and customize user registration and authentication features into the site, and create a restricted area of the site accessible only to authenticated users.

[4] http://wedewlawns.com
[5] https://mandrill.com/

Chapter 3. Integrating Stripe and Creating a Product Catalog

With the site launched and the mailing list expanding, Mr. McDew's ambitions have now turned to online sales. He would like to create a simple online store for selling products such as lawn mowers and gardening tools. In this chapter you'll integrate Stripe and Laravel Cashier[6] to easily and securely accept credit card payments. You'll also learn how to create a restricted administration console complete with a product catalog manager so Mr. McDew and his team can easily add and update products.

Chapter 4. Selling Electronic and Physical Products

With Stripe and Laravel Cashier integrated, and your product catalog created, it's time to begin selling products! In this chapter you'll learn how to sell both electronic (downloadable) and physical products through your website. We'll show you how to integrate the secure Stripe "Buy" button and modal, complete transactions using Cashier, and generate one-time URLs for automated product downloads. We'll also talk a bit about shipping and sales tax complexities, and identify a few popular associated resources.

Chapter 5. Selling Lawn Care Service Subscriptions

In this project phase, Mr. McDew sets his sights on selling lawn care subscriptions. In this chapter you'll learn how to integrate Stripe's recurring billing service into the We Dew Lawns website, and sell a number of lawn care service tiers to area customers. You'll also learn how to configure webhooks to autonomously receive and respond to various subscription-related events.

Chapter 6. Integrating a Shopping Cart

While the site is operational and customers can purchase a variety of products and services, the biggest drawback is the lack of a shopping cart which prevents customers from purchasing multiple products at once. This outcome wasn't accidental, because we've preferred to focus on fundamental Laravel- and Stripe-related features so as to ensure everything is working flawlessly before introducing more complicated capabilities. In this chapter we'll change that by adding a shopping cart to the site. By the conclusion of this chapter you'll have successfully integrated a shopping cart into the site, allowing customers to truly shop the site, and conveniently purchase more than one product during a single session.

Downloading the We Dew Lawns Project

The very same code used to power the WeDewLawns.com website is included with your Leanpub purchase. Be sure to carefully read the `README` for installation instructions. As we inevitably fix bugs and improve the code, we'll distribute updates through Leanpub and notify you whenever significant improvements are available.

[6] http://laravel.com/docs/master/billing

About the Authors

Authors Eric L. Barnes and W. Jason Gilmore had a lot of fun collaborating on this book over several months. They are both well-regarded figures in the PHP and Laravel Community and have more than 30 years of combined experience working with the PHP language.

Eric L. Barnes

Eric L. Barnes[7] runs and operates Laravel News[8], a site bringing the Laravel community all the latest news and information about the framework, and is a full time product engineer. He has worked with PHP for the past 14 years.

Eric has been writing almost daily on both the Laravel News site and newsletter since January, 2014 and he loves both the Laravel community and ecosystem. In his spare time he enjoys his family, playing golf, and inspires to constantly be learning new things.

W. Jason Gilmore

W. Jason Gilmore[9] is author of the bestselling book, Easy Laravel 5[10]. He's a software developer and consultant who has spent much of the past 17 years helping companies of all sizes build amazing solutions. Recent projects include a Stripe-powered SaaS for the interior design industry, an e-commerce analytics application for a globally recognized publisher, a Linux-powered autonomous environmental monitoring buoy, and a 10,000+ product online store.

Over the years Jason has authored eight books on web development, published more than 300 articles within popular publications such as Developer.com, JSMag, and Linux Magazine, and instructed hundreds of students in the United States and Europe. Jason is co-founder of the wildly popular CodeMash Conference[11], the largest multi-day developer event in the Midwest.

Away from the keyboard, you'll often find Jason hanging out with his family, hunched over a chess board, and having fun with DIY electronics.

Contact the Authors

Nobody is perfect, particularly when it comes to writing about technology. We've probably made a few mistakes in both code and grammar, and strive to correct issues almost immediately upon notification. If you would like to report an error, ask a question or offer a suggestion, please e-mail us at support@easyecommercebook.com.

[7] http://ericlbarnes.com
[8] https://laravel-news.com
[9] http://www.wjgilmore.com
[10] http://easylaravelbook.com
[11] http://www.codemash.org

Chapter 1. Creating the We Dew Lawns Website

In this opening phase you've been tasked with creating what effectively amounts to brochureware, albeit one backed by the powerful Laravel framework which sets the stage for more exciting features in later chapters. Presumably you're already capable of creating such a site, and indeed the goal of this chapter isn't to show you how to create a controller or view. Instead, the goal is to methodically implement the WeDewLawns.com in a way that mimics what you might do when working with a real-world project, later project phases won't require significant refactoring or a fundamental rethinking of your developmental approach.

We'll kick things off by creating a new Laravel project and customizing the project layout. You'll also integrate the Bootstrap framework and configure Laravel Elixir to autonomously compile your CSS customizations. From there we'll create a few customary static pages and then create a contact form which uses Laravel 5's new form request feature. We'll integrate the Mandrill[12] e-mail delivery provider in order to ensure efficient and flawless delivery of customer inquiries. We'll wrap things up with an exit interview with company owner Mr. McDew, so pay close attention to the topics discussed throughout the chapter!

Even if you're an experienced Laravel developer we encourage you to follow along all the same because several features new to Laravel 5 are introduced in this chapter.

Creating the We Dew Lawns Project

Let's kick things off by creating a new Laravel project. Presumably you've already installed Laravel using Composer, however if not execute the following command:

```
1 $ composer global require "laravel/installer=~1.1"
```

Once installed you can create the We Dew Lawns project using the `laravel new` command:

[12] https://www.mandrill.com/

```
$ laravel new dev.wedewlawns.com
Crafting application...
Application ready! Build something amazing.
```

With the project created, enter the directory and start PHP's built-in web server using Artisan's `serve` command:

```
$ php artisan serve
Laravel development server started on http://localhost:8000
```

Obviously you're free to us Homestead or another solution for running the application locally. Once the server is up and running head on over to `http://localhost:8000` (or whatever URL you've configured if using Homestead or another solution) to access the default Laravel project home page.

Creating the Project Layout

One of the first steps we like to take whenever starting a new Laravel project is configure the layout file which will encapsulate the various website views. In Laravel 5.1 the example layout file was unfortunately removed, however creating a new one is very easy to do. Create a file named `app.blade.php` inside the directory `resources/views`, and add the following contents to it:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width,
    initial-scale=1">
  <title>Welcome to We Dew Lawns</title>
</head>
<body>
  <h1>
    Welcome to Your Professional Lawn Care Service Provider
  </h1>
  @yield('content')
</body>
</html>
```

This is just a simple layout which we'll use for demonstration purposes; ultimately your layout could (and likely will) differ significantly from this, and that's fine. What's important for you to understand is that the `@yield` directive serves as a placeholder for content which will subsequently be supplied by a view. Any view identified as being associated with this layout will contain a section named `content` which will be injected into this location. But chances are you already know that, so we won't further belabor the point. Instead let's focus on the key syntax used in this layout:

- `<!DOCTYPE html>`: This is the HTML5 document type declaration. It tells the browser to render the page in accordance with any available HTML5 specifications.
- `<html lang="en">`: The `html` tag is an HTML document's root element, and all other elements found in the document should be nested within. The `lang` attribute identifies the natural language used within the page. This is useful for a number of reasons, including font rendering, search engine categorization, and for text readers.
- `<meta charset="utf-8">`: This defines the document's character set. Because English will be used throughout the project website, this isn't such a critical matter to understand however if your particular project ever involved multiple languages then you'll certainly want to spend some time reading up on the importance of character set encodings.
- `<meta name="viewport"`: The `viewport` meta tag was first introduced in Mobile Safari as a solution for allowing developers to determine the size and scale of the page as rendered to the browser screen. This particular definition dictates that the page consume all available space (`width=device-width`) and to not be zoomed in or out in any fashion (`initial-scale=1`). Although the `viewport` tag is not part of any web standard, it is supported by all other mainstream browsers to varying degrees.

There are of course plenty of other ways to initialize the page skeleton however this should be suffice to get the project rolling. Regardless, this layout is hardly appealing, so let's incorporate the Bootstrap framework and use a few of Bootstrap's HTML widgets to create a more compelling interface. We'll do so using a powerful new tool available to Laravel 5 called Elixir, which can not only greatly reduce the amount of work required to maintain your project CSS, but additionally can automate away all sorts of other workflow annoyances.

Introducing Elixir

Elixir[13] is a feature new to Laravel 5, introduced with the purpose of automating many of the otherwise manual tasks associated with modern web development. Among other things you can compile Sass files into CSS, compile CoffeeScript[14] into JavaScript, and execute your PHPUnit/PHPSpec tests. Further, you can configure Elixir to do this *automatically*, meaning every time you save a modified Sass or CoffeeScript file, those changes will automatically be compiled and made available to your application!

To take advantage of Elixir you'll need to install Node.js[15]. Fortunately, installing Node.js is easy no matter your operating system; head on over to http://nodejs.org/[16] for more instructions.

With Node installed, it's time to install Bootstrap. This is done by executing the following command:

[13] https://github.com/laravel/elixir
[14] http://coffeescript.org/
[15] http://nodejs.org/
[16] http://nodejs.org/

```
$ npm install
```

This command will look to a file named `package.json` in order to determine what Node dependencies you'd like to install. This file is included by default inside the root directory of all new Laravel applications. If you're not familiar with the role of this file, just keep in mind that the `package.json` file is to npm what `composer.json` is to Composer. At the time of this writing, that file looked like this:

```
{
  "private": true,
  "devDependencies": {
    "gulp": "^3.8.8"
  },
  "dependencies": {
    "laravel-elixir": "^3.0.0",
    "bootstrap-sass": "^3.0.0"
  }
}
```

As you can see, `laravel-elixir` is identified alongside `bootstrap-sass` as one of the project dependencies. Additionally, a package called `gulp` is identified as a developmental dependency. This is required because Elixir is based atop Gulp, and in fact you'll run your Elixir tasks by way of the `gulp` command. Next you'll learn how to use Elixir to conveniently compile your project's Sass CSS markup as well as integrate and override Bootstrap CSS framework styling.

Integrating Sass and Bootstrap

In just a few short years since the first public initial release in 2011, the Bootstrap[17] framework has become an incredibly popular solution for designing eye-appealing and responsive web applications. It's free, open source, and under constant development, making it an ideal solution for the We Dew Lawns website particularly given the limited design budget.

The early Laravel 5 releases originally included the Less[18]-formatted Bootstrap source files, however this default was changed in later versions. As of Laravel 5.1, Sass[19] is now the default CSS preprocessor, and you can install the Sass-based Bootstrap source files via npm. In fact, when you ran `npm install` to install Elixir, the relevant Bootstrap files were also installed!

Incidentally, in case you were wondering where exactly the Bootstrap files (and other dependency files) were stored, you'll find them inside `node_modules`. However, you *will not* directly modify these files in order to override any default settings! Instead, we'll reference the Bootstrap directory

[17] http://getbootstrap.com/
[18] http://lesscss.org/
[19] http://sass-lang.com/

within a Sass file, override a few defaults, and then compile the Sass file to produce the desired CSS. If you're not familiar with how npm and the `node_modules` directory work, please take some time to review the npm documentation[20].

Because Laravel supports the Sass CSS preprocessor by default, so we'll use that to manage the WeDewLawns project CSS. Sass extends the CSS language in ways that can dramatically improve maintainability and save sanity. When coupled with Bootstrap, developers have at their disposal an amazingly powerful layout and styling solution. We'll use Sass and CSS to manage various stylistic and navigational aspects of the project. Laravel provides you with a default Bootstrap-integrated Sass file, which is located at `resources/assets/sass/app.scss`. It currently contains a single line:

```
1 // @import "node_modules/bootstrap-sass/assets/ \
2    stylesheets/bootstrap";
```

With Bootstrap having been successfully installed using npm, you can now uncomment this line, removing the preceding `//` from `@import`. This `@import` statement is a CSS feature, and it imports all of the Bootstrap source files. Take careful note that we referred to these files as *source files*, because they aren't the actual CSS files you would include in your project web page. You will compile these files into the final CSS, and optionally override the default styling. To override the styling you'll use Sass syntax, so if you'd for instance like to override Bootstrap's primary brand styling to use the color `#597420` instead of the default `#337ab7`, you'll make the following declaration in `app.scss`:

```
1 $brand-primary: #597420
```

Of course, you'll place this line below the `@import` statement so that Bootstrap's initial declaration will be subsequently overridden.

You can view a complete list of available Bootstrap Sass variables by perusing the bootstrap-sass source[21].

Remember that `app.scss` is just your CSS source file, meaning you'll need to compile it into its final form before referencing it within the project's HTML layout. There are several ways to do this, however *Elixir* was built precisely for this purpose!

Compiling Your Sass Source Files

Open the `gulpfile.js` file found in the project root directory and you'll see the following contents:

[20] https://docs.npmjs.com/

[21] https://github.com/twbs/bootstrap-sass/blob/master/assets/stylesheets/bootstrap/_variables.scss

```
var elixir = require('laravel-elixir');

elixir(function(mix) {
    mix.sass('app.scss');
});
```

That call to `mix.sass` is a Gulp task provided to you by Elixir, and it's responsible for compiling `app.scss` and placing the compiled results into `public/css/app.css`. Thanks to Elixir you don't have to worry about identifying the source nor the destination directories; you can just identify the Less file as demonstrated in this example, and run the following command to compile it:

```
$ gulp
[18:38:23] Using gulpfile ~/Software/dev.wedewlawns.com
  /gulpfile.js
[18:38:23] Starting 'default'...
[18:38:23] Starting 'sass'...

Fetching Sass Source Files...
   - resources/assets/sass/app.scss

Saving To...
   - public/css/app.css

[18:38:24] Finished 'default' after 712 ms
[18:38:24] gulp-notify: [Laravel Elixir] Sass Compiled!
[18:38:24] Finished 'sass' after 811 ms
```

Executing `gulp` resulted in `app.scss` being compiled and the results placed into `public/css/app.css`. This means that because `app.scss` is importing the Bootstrap source files, the resulting `app.css` file will be relatively large. Of course, by minifying the resulting CSS you'll dramatically reduce the overall file size (something else Elixir can automate for you). With the `app.css` file now available, open the `resources/views/app.blade.php` and include the compiled `app.css` file within the layout header:

```
...
<head>
  <meta charset="UTF-8">
  <title>Welcome to We Dew Lawns</title>
  <link rel="stylesheet" href="/css/app.css">
</head>
...
```

With the `app.css` file in place, all of the Bootstrap stylings are available for use within the application!

Of course, repeatedly returning to the terminal to execute `gulp` is going to get old fast. Fortunately you can direct Gulp to listen for changes and then automatically recompile the files for you by running the following command:

```
$ gulp watch
```

Using the Bootstrap jQuery Components

After integrating Bootstrap into the Sass asset file you'll likely want to incorporate jQuery[22] in order to take advantage of Bootstrap's various jQuery plugins. Even if you don't plan on using these plugins you'll nonetheless want to integrate jQuery because we'll use the library in Chapter 5 to implement part of the recurring payments feature. There are several ways to include jQuery however the easiest involves using one of several CDNs such as that one provided by jQuery:

```
<script src="//code.jquery.com/jquery-1.11.3.min.js"></script>
```

Alternatively you can create an Elixir task to copy the jQuery files found in the npm Bootstrap installation. See the Elixir documentation[23] for more information about other Elixir capabilities.

Creating the Home Page

All new Laravel projects include a default home page which renders the output found in the below screenshot.

[22] http://jquery.com
[23] http://laravel.com/docs/master/elixir

The Home Page

Let's start to customize the site to suit the We Dew Lawns project. Open `resources/views/welcome.blade.php` and replace the contents with the following:

```
@extends('app')

@section('content')
  <h1>Welcome to Your Home Lawn Care Provider</h1>
@endsection
```

This is of course an incredibly simple home page, intended simply to replace the boilerplate provided by a default Laravel 5 application. As you can see on http://wedewlawns.com[24], the actual home page is somewhat more sophisticated, so be sure to have a look at the companion source code.

Creating the About Us Page

A family owned-and-operated business spanning multiple generations, We Dew Lawns has built quite a reputation for friendly and timely service. They would like to highlight the company history on the site, so let's create a new controller named `About` and use the `index` action and corresponding view to house this information:

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

Because Laravel by default creates a RESTful (also known as a *resource*) controller, the `AboutController.php` class will contain seven actions, including `index`, `show`, `create`, `store`, `edit`, `update`, and `destroy`. For the purposes of serving a standard "About Us" page we'll only use the `index` action, meaning you can remove the other six. Modify `app/Http/Controllers/AboutController.php` to look like this:

[24] http://wedewlawns.com

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use App\Http\Requests;
use App\Http\Controllers\Controller;

class AboutController extends Controller {

  public function index()
  {
    //
  }

}
```

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

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

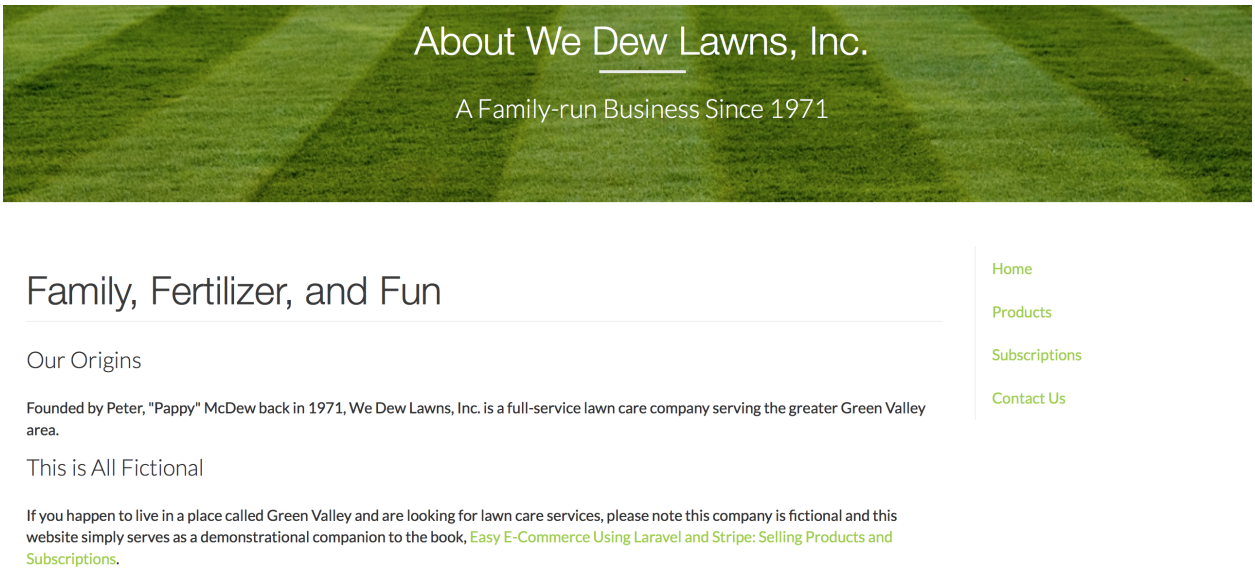
This tells Laravel to return a view named `index.blade.php` which resides in the directory `resources/views/about`. Because the `about` directory doesn't yet exist, go ahead and create it now. Inside this directory create a file named `index.blade.php`, adding the following contents to it:

```
@extends('app')

@section('content')

<h1>About We Dew Lawns, Inc.</h1>

<p>
Welcome to We Dew Lawns, a family-run lawn care company
serving the greater Green Valley area.
</p>

@endsection
```

Finally, open `app/Http/routes.php` and add the following line:

```
1 Route::resource('about', 'AboutController', ['only' => ['index']]);
```

Save your changes and with the route definition in place, head over to `http://localhost:8000/about` and you should see the new page. The example found in the company website is a tad more involved (see below screenshot), but the idea remains the same.



The About Us page

Adding a Contact Form

Todd McDew spends quite a bit of time fielding questions from prospective customers, and would like to streamline a bit of the discussion by adding a user-friendly contact form to the site. Any information submitted through this form will be sent via e-mail to the Ms. Organo, who will gather the requested information and return it to the potential customer via their desired mode of correspondence (e-mail or telephone).

In this section you'll learn how to create a contact form and process it using Laravel 5's new Form Request feature. You'll also learn how to integrate the Mandrill[25] e-mail delivery service to ensure e-mails are reliably delivered.

Introducing Mandrill

Mandrill[26] is an e-mail delivery service managed by the same company behind MailChimp[27]. Mandrill eliminates most of the hassle you'd otherwise likely encounter when attempting to manage

[25] https://mandrill.com/
[26] https://mandrill.com/
[27] http://www.mailchimp.com

mail delivery yourself, and offers features useful for ensuring your e-mails aren't misidentified as spam. Further, Mandrill won't squeeze the company pocketbook, since the service is free for up to 12,000 e-mails per month.

While Mandrill might seem like overkill if your only intent is to send submitted contact form data to a designated address, keep in mind e-mail delivery has become an increasingly complex matter due to the array of spam prevention measures instituted in recent years. While the contact form data will always be submitted to the same company address, in later project phases you'll need to deliver e-mail to various customer e-mail addresses for a variety of other reasons, including account password recovery. You'll want to be certain these sort of e-mails have the highest likelihood of not getting caught up in filters, and having it delivered by a reliable and well-known delivery service such as Mandrill is going to improve those chances.

You're certainly not compelled to use Mandrill, although the authors like it. Any of Laravel's supported solutions are valid, however some come with a certain level of overhead therefore be prepared to do some additional work if you don't plan on using a third-party mail delivery service.

Integrating Mandrill

To integrate Mandrill, you'll first need to create a new account over at https://mandrill.com/[28]. After doing so, you'll need to complete a few simple configuration-related tasks. To begin, you'll need to install the Guzzle[29] HTTP library. Open up `composer.json` and add the package:

```
"require": {
  ...
  "guzzlehttp/guzzle": "~5.3|~6.0"
},
```

After saving the changes, install the package:

```
$ composer install
```

Next, login to your Mandrill account and retrieve your secret API key. You'll find it by clicking on the `Settings` icon and reviewing the section titled "API Keys". You'll also want to take note of the Mandrill SMTP host (`smtp.mandrillapp.com`), port (`587`), and your SMTP username (your e-mail address).

Open your `.env` file and modify your `MAIL_*` configuration parameters to use the following information:

[28] https://mandrill.com/

[29] https://github.com/guzzle/guzzle

```
MAIL_DRIVER=mandrill
MAIL_HOST=smtp.mandrillapp.com
MAIL_PORT=587
MAIL_FROM=YOUR_MANDRILL_EMAIL_HERE
MAIL_NAME=EMAIL_FROM_NAME
MAIL_USERNAME=YOUR_MANDRILL_USERNAME_HERE
MAIL_PASSWORD=YOUR_SECRET_API_KEY_HERE
```

Save those changes and update the `config/services.php` file to identify the API KEY:

```
'mandrill' => [
  'secret' => env('MAIL_PASSWORD'),
],
```

Finally, open `config/mail.php` and update the `from` array key like so:

```
'from' => ['address' => env('MAIL_FROM'), 'name' => env('MAIL_NAME')],
```

Save these changes and you're ready to begin sending e-mail through Mandrill!

Creating the Form

Let's create a new RESTful controller called `Contact` which will house the logic used to present and process the contact form:

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

You'll find the `Contact` controller in `app/Http/Controllers/ContactController.php`. We'll only use the `create` and `store` actions to present and process the contact form, meaning you can delete the other actions. After doing so, update the `routes.php` file so Laravel knows about the newly created controller:

```
Route::resource('contact', 'ContactController',
  ['only' => ['create', 'store']]);
```

Next, return to `ContactController.php` and modify the `create` action to look like this:

```
public function create()
{
    return view('contact.create');
}
```

Next we'll create the contact form. You can hand-code the HTML or could optionally take advantage of the `laravelcollective\HTML` package, which among other things includes a convenient `Form` facade. This facade offers a series of concise methods useful for quickly constructing forms. For instance the following two statements produce identical form labels:

```
<label for="name">Your Name</label>

{!! Form::label('name', 'Your Name') !!}
```

If you haven't used a facade such as `Form` before, you might be wondering what one has to gain as compared to just manually writing out the tags yourself. The primary reason is consistency; when writing the tag manually you're free to place the various tag attributes in any order you please, which often leads to chaotic code. When using the facade you're required to follow a more rigorous ordering, making subsequent modifications and debugging much easier.

If you'd like to use the facade (we like it and will use it throughout most of the book), keep in mind this package was previously part of Laravel however beginning with the version 5 release you'll need to manually add it via Composer. Open `composer.json` and add the following line to the `require` section:

```
"require": {
  ...
  "laravelcollective/html": "5.1.*"
},
```

Save the changes and run `composer update` to install the package. Next, add the following line to the `providers` array found in your `config/app.php` array:

```
Collective\Html\HtmlServiceProvider::class,
```

Don't close the file just yet because you'll also need to add the following line to the `config/app.php` file's `aliases` array:

```
'Form' => Collective\Html\FormFacade::class,
'Html' => Collective\Html\HtmlFacade::class
```

With these two new lines in place, save the changes and you'll be able to take advantage of the `Form` facade. Create a new directory named `contact` inside `resources/views` and inside it create a file named `create.blade.php`. Add the following contents to this file:

```
@extends('app')

@section('content')

<h1>Contact WeDewLawns</h1>

@if (count($errors) > 0)
    <div class="alert alert-danger">
        Please correct the following errors:<br />
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif

{!! Form::open(array('route' => 'contact.store',
  'class' => 'form', 'novalidate' => 'novalidate')) !!}

<div class="form-group">
    {!! Form::label('Your Name') !!}
    {!! Form::text('name', null,
        array('required',
              'class'=>'form-control',
              'placeholder'=>'Your name')) !!}
</div>

<div class="form-group">
    {!! Form::label('Your E-mail Address') !!}
    {!! Form::text('email', null,
        array('required',
              'class'=>'form-control',
              'placeholder'=>'Your e-mail address')) !!}
</div>

<div class="form-group">
    {!! Form::label('Your Message') !!}
    {!! Form::textarea('message', null,
        array('required',
              'class'=>'form-control',
              'placeholder'=>'Your message')) !!}
```

```
43  </div>
44
45  <div class="form-group">
46      {!! Form::submit('Contact Us!',
47        array('class'=>'btn btn-primary')) !!}
48  </div>
49  {!! Form::close() !!}
50
51  @endsection
```

Save these changes and navigate to `/contact/create` and you should see the form presented in the following screenshot.

The We Dew Lawns contact form

Of course, strictly adhering to the REST routing norms in this case is weird, because it would be more convenient and natural to access the contact form via `/contact` so let's remove the `Contact` controller's resourceful route definition in `app/Http/routes.php`, replacing it with the following two definitions:

```
Route::get('contact',
  ['as' => 'contact',
   'uses' => 'ContactController@create']);
Route::post('contact',
  ['as' => 'contact_store',
   'uses' => 'ContactController@store']);
```

You'll also need to update the form's `route` setting, changing it from `contact.store` to `contact_store`.

With these changes saved you should be able to access the contact form via `http://localhost:8000/contact`. Next we'll implement the `Contact` controller's `store` method, in addition to the form request used to validate the form data.

Creating the Contact Form Request

We'll take advantage of Laravel 5's new *form request* feature to formalize the validation of data submitted through the contact form. To generate a form request, execute the following command:

```
$ php artisan make:request ContactFormRequest
Request created successfully.
```

The newly generated request was saved to a file named `ContactFormRequest.php` and placed within the directory `app/Http/Requests`. Open it up in your editor and you'll see the following class (comments removed):

```
<?php

namespace App\Http\Requests;

use App\Http\Requests\Request;

class ContactFormRequest extends Request {

  public function authorize()
  {
    return false;
  }

  public function rules()
  {
    return [
```

```
        //
      ];
    }

}
```

The `authorize` method can optionally be used to verify whether the submitting user is authorized to submit form data. Because any user should be able to use this form you should modify the method to return `true` instead of `false`:

```
public function authorize()
{
  return true;
}
```

The `rules` method is responsible for validating the submitted form data. To do so you'll define an associative array within the `return` statement, associating each form field name with a validation declaration. For instance the `name` and `message` fields are required, so you'll associate each with the `required` validator. The `email` validation requirement is a bit more complicated, because not only is it required but it should be syntactically valid. In order to suit these requirements, update the `rules` method to look like this:

```
public function rules()
{
  return [
    'name' => 'required',
    'email' => 'required|email',
    'message' => 'required',
  ];
}
```

Processing the Contact Form

With the form request in place, all that remains is to update the `Controller` action's `store` action. This action will e-mail the inquiring user's contact information and correspondence to We Dew Lawns' assistant Patty Organo. Open the `Contact` controller and navigate to the `store` action, updating it to look like this:

```
use App\Http\Requests\ContactFormRequest;

...

public function store(ContactFormRequest $request)
{
    $data = [
        'name' => $request->get('name'),
        'email' => $request->get('email'),
        'user_message' => $request->get('message'),
    ];

    \Mail::send('emails.contact', $data, function($message)
    {
        $message->from(env('MAIL_FROM'));
        $message->to(env('MAIL_FROM'), env('MAIL_NAME'));
        $message->subject('WeDewLawns.com Inquiry');
    });

    return \Redirect::route('contact')
      ->with('message', 'Thanks for contacting us!');
}
```

Next you'll want to create the e-mail message view. A directory for managing e-mail views is already found in `resources/views/emails`. This directory was created by default because Laravel uses the directory to manage the password recovery e-mail view. Create a new file in this directory named `contact.blade.php` and add the following contents to it:

```
<p>
A prospective customer named {{ $name }} ({{ $email }})
has submitted an inquiry through WeDewLawns.com:
</p>

<p>
{{ $user_message }}
</p>
```

Deploying the Website

There are countless different ways in which you can deploy a Laravel application, and covering one or even a few of them here isn't going to be practical because the instructions would only apply to a

small segment of the readership. We've had luck deploying Laravel applications using Capistrano[30] to a number of different hosting providers, including DreamHost and DigitalOcean, and have used Heroku's excellent deployment tooling with great success.

If you're looking for a turnkey deployment solution, check out Forge[31]. Forge was created and is managed by Taylor Otwell[32], who also happens to be the creator of Laravel. Also check out Envoyer[33] for another deployment solution also created and maintained by Taylor Otwell.

The Project Presentation

The first project phase has been successfully deployed, and the time has come for you and Mr. McDew to discuss progress. You've just arrived at the company headquarters, and Ms. Organo has ushered you into the conference room. After some small talk she warns you that Mr. McDew seems to be in a particularly surly mood this morning. She's muttering something about a broken lawn mower when you see Mr. McDew's monster truck veer into the parking lot. He climbs out and slams the door shut, spilling coffee all over himself in the process. This meeting might not be pretty. A moment later Mr. McDew enters the office, settles into his desk chair, and puts his feet up.

Todd McDew (TM): Sooooo, Mr. Web Developer (Mr. McDew has never actually called you by your name, instead choosing to call you variations of Mr. Computer, Mr. Programmer, and other similarly creative monikers), I see the first phase seems to have been completed. It looks good, but I have a few questions.

Mr. Web Developer (that's you): OK great sir, I'm happy to…

TM: Zip it son. I haven't asked a question yet.

MWD: Sorry, sir.

TM: I sent twenty seven e-mails through the website contact form last night, and Patty told me they all arrived in her inbox. How much did those e-mails cost me?

MWD: The site uses a third-party mail delivery service called Mandrill. You can send up to 12,000 emails per month without charge. Also, the service is entirely managed separately from the website, meaning you likely won't have to spend extra money and time troubleshooting technical issues.

TM: The contact form looked great on both my mobile phone and laptop. Most websites I see look horrible on a mobile phone. How did you accomplish this?

MWD: We use a popular and free tool called Bootstrap for creating what are called responsive web layouts. It works equally well on mobile phones, tablets, and computers, meaning users will always have a friendly experience.

[30] http://capistranorb.com/

[31] https://forge.laravel.com/

[32] https://twitter.com/taylorotwell

[33] https://envoyer.io/

TM: You've answered all of my questions, and I've got to get back to fixing that lawn mower, so you're hired for Phase 2. In this next phase I'd like you to start collecting customer e-mail addresses. Update the site to let users create an account. We can provide anybody who registers with printable coupons for use with our lawn services, but don't just let any whippersnapper download these coupons! They must only be available to registered users.

MWD: OK great, thank you si…

TM: Zip it son, that wasn't a question.

Onwards to Phase 2!

Summary

In this opening chapter you've successfully created and deployed the first project phase, however there is plenty more work to do. In the next chapter you'll integrate a user registration and authentication system, and restrict part of the site to authenticated users.

Chapter 2. Integrating User Accounts

Mr. McDew's ultimate goal is to dramatically increase his company's revenue by expanding into the online sale of products and lawn care services through WeDewLawns.com, but he isn't ready to make that leap just yet. After having deployed the initial site iteration, he's now intent on building a mailing list. His plan is to offer special discounts to users who register on the We Dew Lawns website, meaning your task is to add user registration and authentication capabilities to the site using Laravel's Auth service, and then create a restricted discount page available only to registered users.

Fortunately, Laravel 5 includes authentication scaffolding which includes facilities for registration, authentication, and page restriction, meaning you can save a substantial amount of time and hassle by taking advantage of these native capabilities. Let's begin by adding account registration to the site.

Integrating User Account Registration

The most logical first step associated with user account integration is implementation of registration capabilities. Any visitor interested in creating an account will need to provide a few details such as his name, e-mail address and a password via a web form. Once validated, this information will be stored in the database.

To get started, we will need to create a table for storing the user information. All new Laravel projects actually come with a migration containing the instructions necessary to create just such a table. To create the table, run the following command:

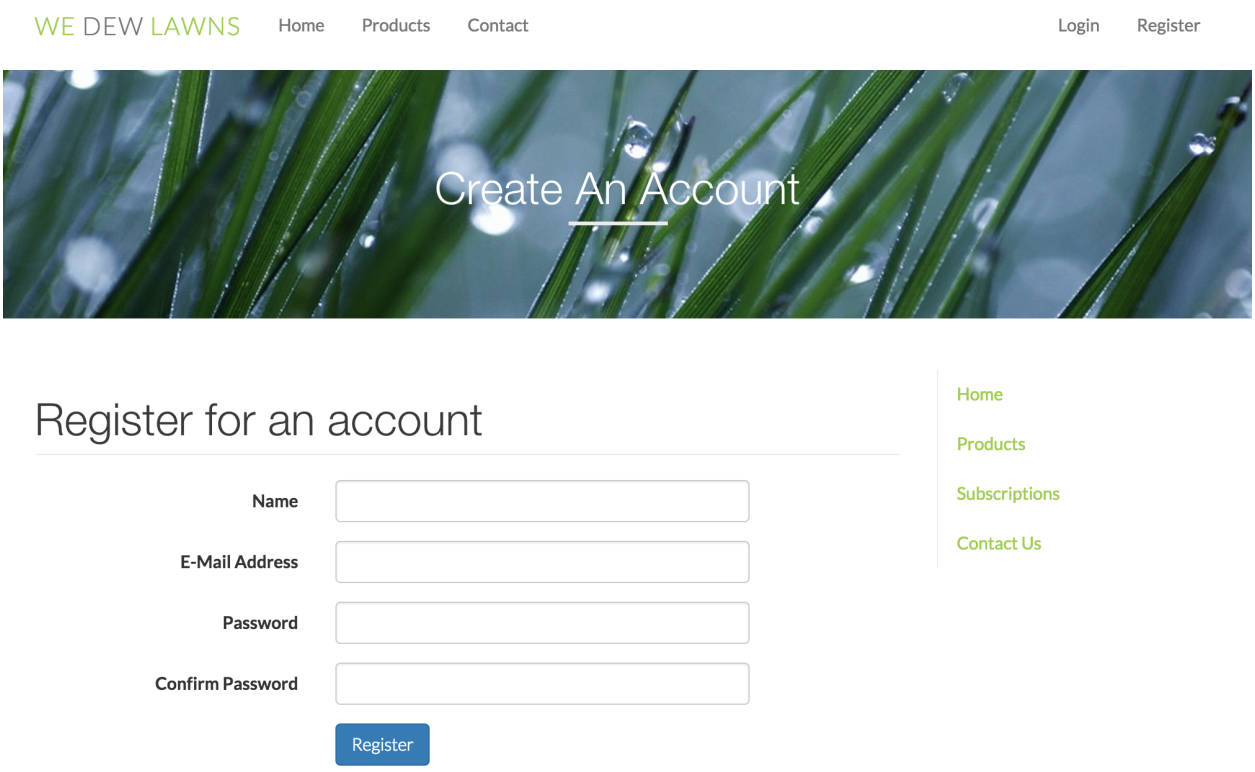
```
1 $ php artisan migrate
2 Migrated: 2014_10_12_000000_create_users_table
3 Migrated: 2014_10_12_100000_create_password_resets_table
```

The `php artisan migrate` command scans the `database/migrations` folder and will execute each file found. These migration files allow you to easily evolve your database structure over time with the added bonus of managing the schema definitions in source control. The two migration files included with every new Laravel project include `2014_10_12_000000_create_users_table.php` which handles the creation of a users table to hold all the details about a registered user, and `2014_10_12_100000_create_password_resets_table.php` which manages information pertaining to password reset requests.

The view files for registration and password resets are not included by default in Laravel 5.1. However, the documentation includes copy and paste code blocks that can be used to create these. To

save you time we have these already setup in the companion project and included small adjustments to offer more advanced styling.

If you navigate to `/auth/register` you'll see the following form:



Customized Registration Form

Now that we've seen the route and the view lets look at the `Auth` Controller which handles actually displaying the view to the browser.

Introducing the Authentication Controller

Now open the `app/Http/Controllers/Auth/AuthController.php`. You'll see this file is pretty sparse, containing only a constructor and a few `use` statements. Here is the file (with the comments stripped out):

```php
<?php

namespace App\Http\Controllers\Auth;

use App\User;
use Validator;
use App\Http\Controllers\Controller;
use Illuminate\Foundation\Auth\ThrottlesLogins;
use Illuminate\Foundation\Auth\AuthenticatesAndRegistersUsers;

class AuthController extends Controller
{
    use AuthenticatesAndRegistersUsers, ThrottlesLogins;

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

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

    protected function create(array $data)
    {
        return User::create([
            'name' => $data['name'],
            'email' => $data['email'],
            'password' => bcrypt($data['password']),
        ]);
    }
}
```

What are Traits?

Traits are groups of methods that can be included in multiple classes. While traits cannot be

> independently instantiated, they provide a convenient way to enhance the behavior of other classes by injecting the trait methods into the class using the `use` statement.

We're interested in the line `use AuthenticatesAndRegistersUsers`. This is a PHP trait which will import all the methods from this class into the `AuthController`. If you would like to learn more about how this works, take a look at the file `AuthenticatesAndRegistersUsers.php` found in the directory `vendor/laravel/framework/src/Illuminate\Foundation\Auth`. This is the class that is imported via the `use` statement. Keep in mind you would never want to edit this file. If you would like to for instance override a method found in this class you can instead create the same method in your controller and customize the logic accordingly.

The two methods for registration are `getRegister()` and `postRegister()`. The `getRegister()` method is responsible for displaying of the template view file, whereas `postRegister()` processes the input, including validating the data and inserting it into the database followed by logging in the user and finally redirecting.

To meet Mr. McDew's requirements we are going to need to modify both the default registration and login features. Let's do so next.

Updating the Registration Fields

Mr. McDew has been adamant about collecting each user's address, city, state, and zip code in order to send targeted emails to people residing in specific areas. To do so we will need to add those columns to the database by creating a new migration, adjusting the registration form, updating the validation, and finally adjust the way users are created to include these new fields.

The Artisan command line tool will create the migration file for us automatically by running the following from terminal:

```
1 $ php artisan make:migration AddUserLocation --table=users
2 Created Migration: 2015_03_26_011808_AddUserLocation
```

If you open this new file, `database/migrations/2015_03_26_011808_AddUserLocation.php`, you will see just an empty class. Let's modify this to include the desired columns:

```
public function up()
{
  Schema::table('users', function(Blueprint $table)
  {
    $table->string('address');
    $table->string('city');
    $table->string('state');
    $table->string('zip');
  });
}

public function down()
{
  Schema::table('users', function(Blueprint $table)
  {
    $table->dropColumn(['address', 'city', 'state', 'zip']);
  });
}
```

Now run the migrations so our database will be up to date:

```
$ php artisan migrate
Migrated: 2015_03_26_011808_AddUserLocation
```

Next we will need to add these new fields to our form. Open `resources/views/auth/register.blade.php` and find the name field:

```
<div class="form-group">
  <label class="col-md-4 control-label">Name</label>
  <div class="col-md-6">
    <input type="text" class="form-control" name="name"
      value="{{ old('name') }}">
  </div>
</div>
```

Just under this add our new fields:

```
<div class="form-group">
  <label class="col-md-4 control-label">Address</label>
  <div class="col-md-6">
    <input type="text" class="form-control" name="address"
      value="{{ old('address') }}">
  </div>
</div>
<div class="form-group">
  <label class="col-md-4 control-label">City</label>
  <div class="col-md-6">
    <input type="text" class="form-control" name="city"
      value="{{ old('city') }}">
  </div>
</div>
<div class="form-group">
  <label class="col-md-4 control-label">State</label>
  <div class="col-md-6">
    <input type="text" class="form-control" name="state"
      value="{{ old('state') }}">
  </div>
</div>
<div class="form-group">
  <label class="col-md-4 control-label">Zip</label>
  <div class="col-md-6">
    <input type="text" class="form-control" name="zip"
      value="{{ old('zip') }}">
  </div>
</div>
```

This form should be pretty straightforward, although you might not be familiar with the `old()` method. This is a convenience method provided by Laravel to easily populate form fields with previously submitted data should the form be rendered anew due to validation errors.

With these fields in place our next step is to adjust the validation to make these fields required. This is a simple change, just open `app/Http/Controllers/Auth/AuthController.php` and you will see a `validator` method:

```
public function validator(array $data)
{
  return Validator::make($data, [
    'name' => 'required|max:255',
    'email' => 'required|email|max:255|unique:users',
    'password' => 'required|confirmed|min:6',
  ]);
}
```

Add the newly created fields to the validator so it looks like this:

```
public function validator(array $data)
{
  return Validator::make($data, [
    'name' => 'required|max:255',
    'address' => 'required',
    'city' => 'required',
    'state' => 'required',
    'zip' => 'required',
    'email' => 'required|email|max:255|unique:users',
    'password' => 'required|confirmed|min:6',
  ]);
}
```

Our final step to implement the user location is to adjust the database insertion method to include these new fields. Locate the `create()` method within the same `app/Http/Controllers/Auth/AuthController.php` file and add the fields:

```
public function create(array $data)
{
  return User::create([
    'name' => $data['name'],
    'email' => $data['email'],
    'address' => $data['address'],
    'city' => $data['city'],
    'state' => $data['state'],
    'zip' => $data['zip'],
    'password' => bcrypt($data['password']),
  ]);
}
```

If you submit the registration form at this point you may notice something odd. Following successful submission you will be redirected to the `/home` route which is handled by a `HomeController` not the `WelcomeController` as you might expect. The reason Laravel does this is to demonstrate a separate logged in page with a "**You are logged in!**" message. This is great for demoing certain framework features but not suited for our project, so let's modify this behavior to ensure users are instead redirected to the proper home page following successful registration.

Open up `app/HTTP/Controllers/Auth/AuthController.php` and we can set a special class property named `$redirectPath` to tell the framework where we want the user to be redirected:

```
use AuthenticatesAndRegistersUsers;

/**
 * Set the redirect path
 * @var string
 */
protected $redirectPath = '/';
```

Now after submitting the registration form you will be redirected to the appropriate home page. With that change we should go ahead and delete the file `app/Http/Controllers/HomeController.php`. Finally open `app/Http/routes.php` and delete this line:

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

With the location fields in place, a new redirect path, and old code removed we are now ready to move on to modifying the sign in.

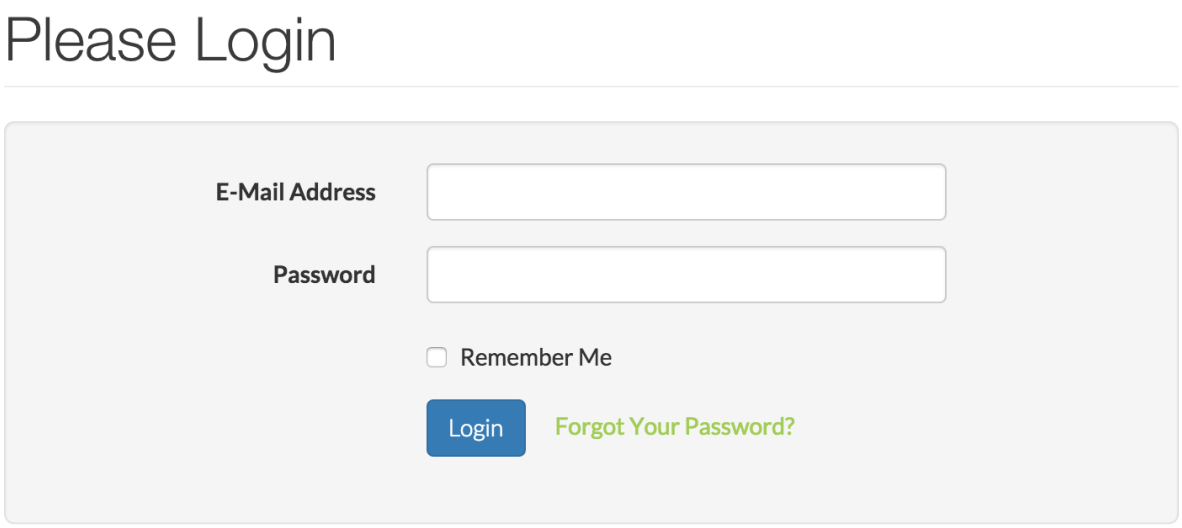
Integrating User Authentication

Registered users will need a way to sign into the website. Like registration, all new Laravel 5 applications include a sign in form which you can see by navigating to `/auth/login`.

Before we move forward let's update this route endpoint to something more concise (from `/auth/login` to `/login`). Open your `app/Http/routes.php` file and add the following route declaration:

```
Route::get('login', [
    'as' => 'login',
    'uses' => 'Auth\AuthController@getLogin',
]);
```

Now we can visit `/login` and see our sign in form:



The Sign in Page

As is the case with registration, Laravel handles this all behind the scenes via the `vendor/laravel/framework/src/Illuminate\Foundation\Auth\AuthenticatesAndRegistersUsers.php` trait. The two methods used for signing in are `getLogin()` and `postLogin()`.

The `getLogin()` method is responsible simply for displaying the sign in form:

```
1  public function getLogin()
2  {
3      return view('auth.login');
4  }
```

As you can see it just returns a view with the template file `auth/login.blade.php`. The real magic happens in the `postLogin()` method:

```
1  public function postLogin(Request $request)
2  {
3    $this->validate($request, [
4      'email' => 'required|email', 'password' => 'required',
5    ]);
6
7    $credentials = $request->only('email', 'password');
8
9    if ($this->auth->attempt($credentials, $request->has('remember')))
```

```
10    {
11      return redirect()->intended($this->redirectPath());
12    }
13
14    return redirect($this->loginPath())
15          ->withInput($request->only('email', 'remember'))
16          ->withErrors([
17            'email' => 'These credentials do not match our records.',
18          ]);
19  }
```

This code first attempts to validate the posted data by ensuring the supplied data includes a valid e-mail address and a password. Next it attempts to sign the user in using the `$this->auth->attempt` method. If it's successful then the user is redirected to the default path that we set in the previous registration section. Finally, if authentication fails it redirects back so you can inform the user their information is invalid.

With this all set it's time to move on to customizing the layout to greet the authenticated user.

Display a Greeting in the Layout Header

Logically we'll want to provide users with an easy visual cue for determining whether they are signed into the website, and if not present registration and sign in links. Typically you'll find these visual cues in the website header. Although you could embed the appropriate logic directly within the application layout, let's organize this in a somewhat more formal fashion by instead managing the logic in a dedicated file. To do so, create a new file named `nav.blade.php`, placing it within the directory `resources/views/inc/`.

This file is known as a *partial* and it's only responsibility will be for displaying the top navigation. By managing it within a separate file you'll be able to easily edit the logic without having to wade through the larger layout file.

You can then place a snippet similar to the following inside this partial:

```
1  @if (Auth::guest())
2      <li><a href="/auth/login">Login</a></li>
3      <li><a href="/auth/register">Register</a></li>
4  @else
5      <li class="dropdown">
6          <a href="#" class="dropdown-toggle" data-toggle="dropdown"
7          role="button" aria-expanded="false">{{ Auth::user()->name }}
8          <span class="caret"></span></a>
9          <ul class="dropdown-menu" role="menu">
```

```
            <li><a href="/auth/logout">Logout</a></li>
        </ul>
    </li>
@endif
```

The snippet uses the `Auth::guest()` static method (included by default within Laravel applications) to determine whether the user is authenticated; if not we'll present the sign in and registration links, otherwise we'll present the user's name and a link to the logout page. It is also worth noting that the `Auth::user()` returns the authenticated user's `User` object, meaning you can use it to retrieve user details such as the e-mail address and name using `Auth::user()->email` and `Auth::user()->name`, respectively.

Let's add a second menu item which will provide authenticated users with an easy way to access the service discounts. Add the following snippet to the above menu:

```
<li class="dropdown">
    <a href="#" class="dropdown-toggle"
       data-toggle="dropdown" role="button"
       aria-expanded="false">{{ Auth::user()->name }}
       <span class="caret"></span></a>
    <ul class="dropdown-menu" role="menu">
        <li><a href="/discounts">Discounts</a></li>
        <li><a href="/auth/logout">Logout</a></li>
    </ul>
</li>
```

Finally, you'll need to reference the partial within the appropriate position of the application layout using the `@include` directive:

```
@include('inc.nav')
```

The next time you load an application page this embedded logic will execute, and the appropriate links and other content will be displayed!

You probably noticed that a second link to a discounts page is presented when the user is authenticated. This points users to the restricted page containing the company's lawn care coupons. Let's implement this feature next.

Creating and Restricting the Discount Controller

Because Mr. McDew's goal is to encourage visitor registration in exchange for lawn care discounts, we'll want to create a page displaying any available coupons which will be accessed at `/discounts`.

This page will display a simple dashboard and tell them about special offers and other "members only" goodies. Because this is a new page we will need to create a controller, setup routes, and finally restrict this to only logged in users.

To get started create a new `Discounts` controller:

```
$ php artisan make:controller DiscountsController --plain
```

Notice that we passed a new argument to the end `--plain`. This tells Laravel that we only want the controller class and to skip generating all the RESTful methods (alternatively you could create a RESTful controller and then delete the unwanted methods). For our purposes we need to just add `index` method which will display our view. Here is an example of the controller:

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

use App\Http\Requests;
use App\Http\Controllers\Controller;

use Illuminate\Http\Request;

class DiscountsController extends Controller {

  public function index()
  {
    return view('member.discounts');
  }

}
```

This returns a view that will be located at `resources/views/member/discount.blade.php`. While Laravel does not require you to manage a controller's views in a dedicated directory, we find it useful for organizational purposes. Next open `routes.php` and add the new route:

```
Route::resource('discounts', 'DiscountsController', ['only' => ['index']]);
```

Because authenticated users will presumably desire to see the latest discounts and other member-specific offers we could optionally automatically redirect authenticated users directly to the restricted page. If you recall from earlier we set the `redirectPath` in `AuthController` to go to the `/` route. Let's change that to our new discount-related route. Open `app/Http/Controllers/Auth/AuthController.php` and override the `$redirectPath` property:

```
class AuthController extends Controller {

    use AuthenticatesAndRegistersUsers;

    public $redirectPath = '/discounts';

    //...
```

With this property in place, the default post-authentication path will be overridden and once a user registers or logs in they will automatically be forwarded here.

Restricting Access to the Discounts Controller

Finally we need to restrict this so only authenticated users can see the page. Laravel provides this capability via *middleware*. If you are not familiar with middleware what it does is provides a convenient mechanism for filtering inbound HTTP requests. Middleware is often intimidating to users not familiar with the concept, however it's really quite simple (and useful); think of middleware as a solution for examining the request before the destination controller action is executed. In doing so you can (among other things) accept or reject the request based on some criteria, and even redirect the user to some other location.

Returning to our controller, Laravel provides two ways to restrict access to it. The first is via the `routes.php` file and the second is through the controller constructor. Either of these methods are fine to use and it comes down your personal preferences. Let's look at how to set it up both ways.

To add it to the route open `app/Http/routes.php` and add this:

```
Route::get('discounts', [
  'middleware' => 'auth',
  'uses' => 'DiscountsController@index'
]);
```

The second way via the controller constructor and is what we decided would be best for this project. Open `app/Http/Controllers/DiscountsController.php` and the following method:

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

Both of these solutions are identical and registers the `auth` middleware and now you can only access any of these methods if you are logged in. So simple, yet so powerful.

As mentioned earlier use the one that looks best to you. Just note that you should pick one or the other. You wouldn't want to set it in both the routes and the controller.

Signing Users Out of the Website

Users will logically desire a way to sign out of their WeDewLawns.com account. The `AuthenticatesAndRegistersUsers` trait handles this for us. If you open the file `Illuminate\Foundation\Auth\AuthenticatesAndR` you'll find the `getLogout` method:

```
1  public function getLogout()
2  {
3      $this->auth->logout();
4
5      return redirect('/');
6  }
```

This method clears any sessions or cookies and redirects the user back to the home page. Because the feature is already implemented all you need to do is create an appropriate route. By default the framework defines the logout URL as `/auth/logout` but let's adjust it to something more succinct. Open your `app/Http/Routes.php` and add the following route:

```
1  Route::get('logout', [
2    'as' => 'logout',
3    'uses' => 'Auth\AuthController@getLogout'
4  ]);
```

Now users will be able to navigate directly to `/logout` in order to sign out of the website.

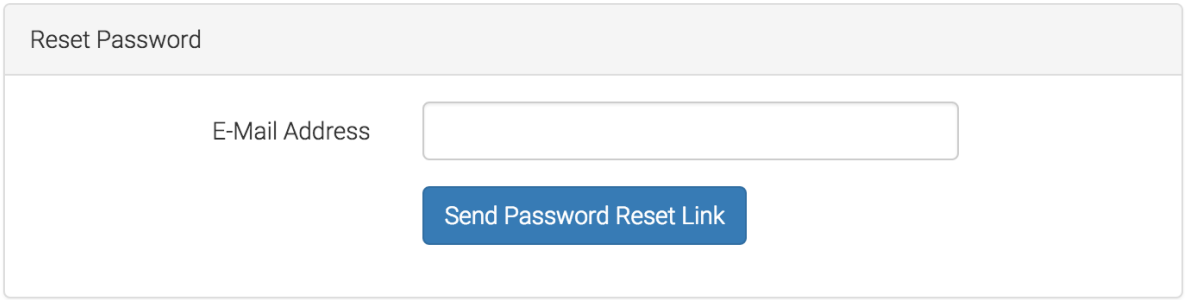
Recovering User Passwords

Registered users will inevitably forget their password, and Mr. McDew certainly doesn't want to be wasting valuable company time on such matters so you'll need a provide users with a simple password recovery solution. Fortunately, like registration and authentication, password recovery is also built into every new Laravel 5 application, meaning there is very little additional work you'll need to do. The controller responsible for presenting and processing password recovery requests is found in `app/Http/Controllers/Auth/PasswordController.php`. Here is the file with all the comments stripped out:

```php
<?php

namespace App\Http\Controllers\Auth;

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

class PasswordController extends Controller
{
    use ResetsPasswords;

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

Just like registration and login this includes a trait to pull in everything needed. The trait is `Illuminate\Foundation\Auth\ResetsPasswords`.

Here is what the default password reset looks like:



Original Password Reset

To make this match the rest of the site you will need to open `resources/views/auth/` folder and edit both the `password.blade.php` and `reset.blade.php`. The password file is used to allow the user to enter their email and be sent a special link which will take them back to the site to **reset** the password.

Mr. McDew fancies himself to be a talented writer, and has instructed us to update the password recovery e-mail using some witty text he's provided. So let's update this e-mail, beginning by opening the `Auth/PasswordController.php` file and customizing the e-mail subject by adding a new class property.

```
$this->subject = 'Reset your password';
```

Next we'll need to customize the e-mail template to include Mr. McDew's quip. Open `views/emails/password.blade.php` and customize it to read like so:

```
It appears you left the sprinklers on too long and your
password was washed away. Not to worry! Follow the link below to
obtain a new password:

{{ url('password/reset/'.$token) }}

Thanks,
Mr. McDew
```

Now when a user forgets the password, this custom message will be delivered to their inbox.

The Project Presentation

The second project phase has been successfully deployed, and now we must meet with Mr. McDew again to discuss the progress with this phase. As you arrive at the We Dew Lawns headquarters, Ms. Organo leads you back into the conference room. Mr. McDew is seated at the head of the table, and seems to be in a particularly nasty mood.

Todd McDew (TM): Since you work for me now I'm going to call you Mr. McDeveloper from now on. Get it? I don't have a lot of time let's get this meeting started. I've been out driving my truck all morning and haven't had a chance to review the site. So let's discuss what you've completed. Are visitors able to register?

Mr. McDeveloper (that's you): Yes sir, they sure can. Would you like me to show you the site in action as we go through this?

TM: No! I don't have time for that. Is the special page available to registered users? Are they able to print coupons?

MWD: Yes it is. I created a special members only section of the site visitors will be forwarded to once they register or login.

TM: Can any whippersnappers access this page without logging in? Will it get indexed by the Google?

MWD: No, thanks to a special feature known as middleware we have a foolproof way to ensure only registered users can access this page. Because it's restricted in this manner search engines will not be able to index the page content and it will be safe from prying eyes.

TM: Middlewhat? Sounds like middlenonsense to me. Anyways, what if users forgot their password? I don't want Patty dealing with this nonsense. Also, did you remember to use my specially prepared remarks to soothe users who forget their password?

MWD: Yes, I set that up. If someone forgets their password they can enter their email address and receive an email with your message and a link to visit and create a new password.

TM: OK. Sounds like you did pretty good on this phase. Let's start the next phase immediately. I want to start selling some of this junk… I mean equipment I have laying around here. I want to start small with only a few products and I want them to pay on the site.

MWD: OK great, thank you si…

TM: I've no time for chit chat son.

Onwards to Phase 3!

Summary

In this chapter we integrated several key features pertaining to user registration and authentication, and provided registered members with a restricted area for perusing special lawn care coupons and other offers. We'll spend the remainder of the book continuing to expand the site's capabilities by building an adminstrative area, integrating a catalog, and selling a variety of lawncare products and subscriptions. We think Mr. McDew is starting to like you, so don't let him down now!

Chapter 3. Integrating Stripe and Creating a Product Catalog

We Dew Lawns' customer base continues to grow thanks to their team's relentless work ethic and the website's extended promotional reach. Mr. McDew's mood even seems to have brightened a bit lately; you even saw him flash a smile during a recent project update meeting. He's also become increasingly emboldened in terms of his ambitions for the company website, now setting his sights on selling lawn and landscaping products to a much larger audience.

In this chapter you'll use the Stripe payment processing service and Laravel Cashier[34] to add a storefront to the website. You'll also create a restricted product catalog manager which the company staff will use to manage electronic and physical products. This sets the stage for the next chapter, in which you'll integrate Stripe's checkout feature into the site so customers can begin purchasing products!

Introducing Stripe

Stripe (https://stripe.com/[35]) is a payment processor founded in 2010 by brothers and software developers John and Patrick Collison. The service was created specifically to ease the historically difficult process of accepting credit card payments through a website. By the authors' estimation the Stripe team has succeeded with flying colors, and in this chapter you'll learn how to integrate Stripe into the website. Before you can do so you'll need to take care of some administrative matters, including creating a Stripe account and getting familiar with the Stripe administrative console.

Creating a Stripe Account

Stripe earns its money by charging a fee of 2.9% + $0.35 per successful transaction, however developers are encouraged to experiment using a free account and the ability to run the service in test mode. To create an account, head over to https://dashboard.stripe.com/register[36]. After providing your e-mail address and password, you'll immediately be logged into the Stripe dashboard. You'll use this dashboard to manage many aspects of your Stripe account.

Although you'll be logged into the Stripe dashboard following registration, you will still need to verify your e-mail address by clicking on the verification link Stripe sends to your inbox. We suggest doing this now before the e-mail gets lost in the shuffle. After verifying your account return to the administration console as we'll spend some time getting acquainted with several key features.

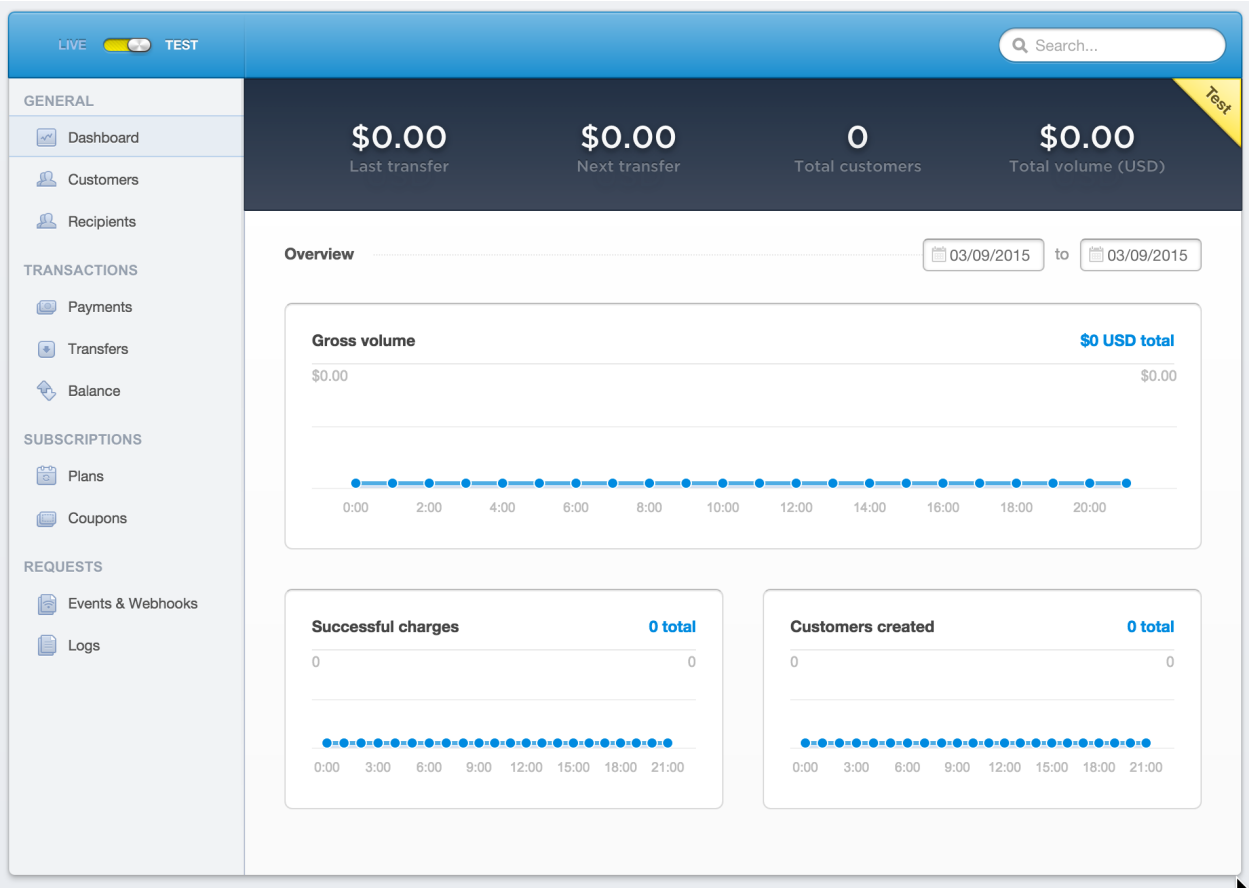
[34] http://laravel.com/docs/master/billing

[35] https://stripe.com/

[36] https://dashboard.stripe.com/register

The Stripe Administration Dashboard

The Stripe administration dashboard offers a central location for managing all matters pertaining to your Stripe account. You'll use this interface to view and manage customers, payments, revenue, configuration settings, and as you'll learn in the next chapter, subscription plans. The interface looks deceptively simple, with less than a dozen menu items located on the left side of the dashboard (see below figure), however once you begin to digging into things you'll find the interface actually contains what are at first a fairly bewildering set of options. In this section we'll save you some time and effort by offering a guided tour of those dashboard features and settings you'll want to understand and possibly modify before beginning the integration process.



The Stripe Dashboard

Your Account Settings

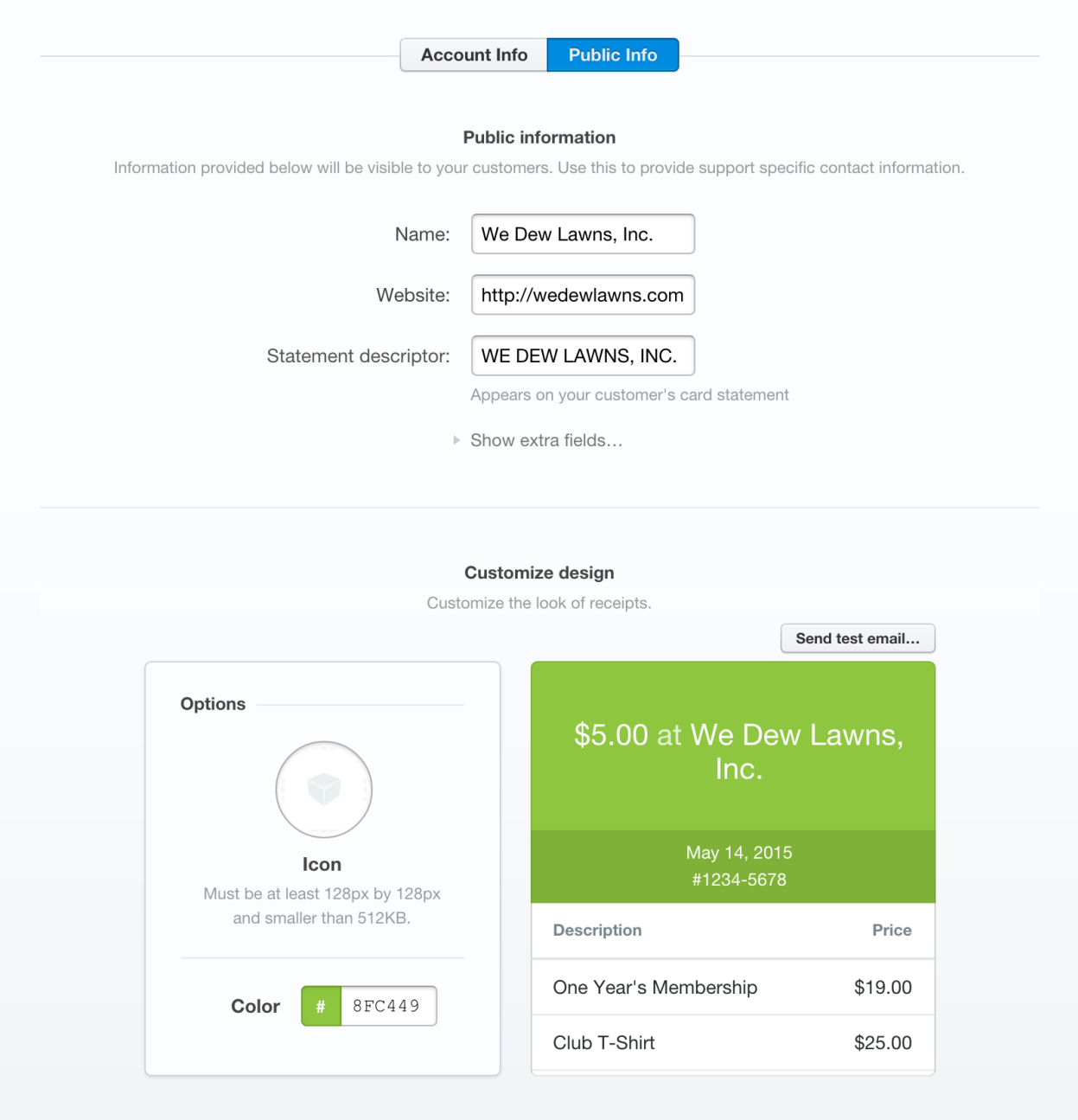
After signing into the dashboard you'll likely want to make a few adjustments to the account settings. Enter the `Account Settings` area by clicking the `Your account` link located at the top left of the dashboard and in the context menu that appears, select `Account Settings`. This interface consists of nine tabs, including:

- **General**: The `General` tab contains settings pertaining to your Stripe account (`Account Info`) and what customers will see on their card statements and in receipts (`Public Info`). We'll talk more about this tab in the section "Updating Your Account and Public Information".
- **Team**: You'll use the `Team` tab to manage team member access to the Stripe administration console. We'll talk more about this tab in the section "Adding Team Members".
- **API Keys**: The `API Keys` tab contains the public and secret keys you'll use in conjunction with the Stripe API. We'll talk more about this tab in the section "Adding Your API Key".
- **Subscriptions**: The `Subscriptions` tab contains settings pertaining to whether and how a subscriber's account should be handled in the event of a failed payment attempt. We'll talk more about this tab in the next chapter.
- **Transfers**: The `Transfers` tab contains settings pertaining to how your company will be paid by Stripe for purchases made through its system. You'll add your bank account information here and additionally tell Stripe how often you'd like to be paid. We'll talk more about this tab in the section "Adding Your Bank Account".
- **Webhooks**: The `Webhooks` tab contains information about how Stripe will communicate with your system in the case of events which occur outside of direct interaction with the API, such as a recurring credit card transaction. We'll talk more about this tab in the next chapter.
- **Connect**: You'll use the `Connect` tab to connect your Stripe account to third party applications such as Shopify[37], IF[38], or Zoho[39].
- **Data**: You'll use the `Data` tab to export your sales data to Quickbooks, delete your test data, or download a sales report. Rather inexplicably this tab also contains the interface you would use to close your Stripe account.
- **Emails**: The `Emails` tab contains the settings used to determine whether you and your customers should be e-mailed for successful payments and refunds.

Don't worry about wading through any of these tabs now although you're encouraged to at least have a cursory look. In the following sections we'll return to many of them in order to adjust key settings.

Updating Your Account and Public Information Head over to the Account Settings' `General` tab and you'll find two additional tabs titled `Account Info` and `Public Info`. The former contains identification-related settings pertaining to your Stripe account while the latter contains settings pertaining to how customers will recognize you on their credit card statements and within e-mail receipts (see below screenshot).

[37] http://shopify.com/
[38] https://ifttt.com/stripe/
[39] https://www.zoho.com/invoice/



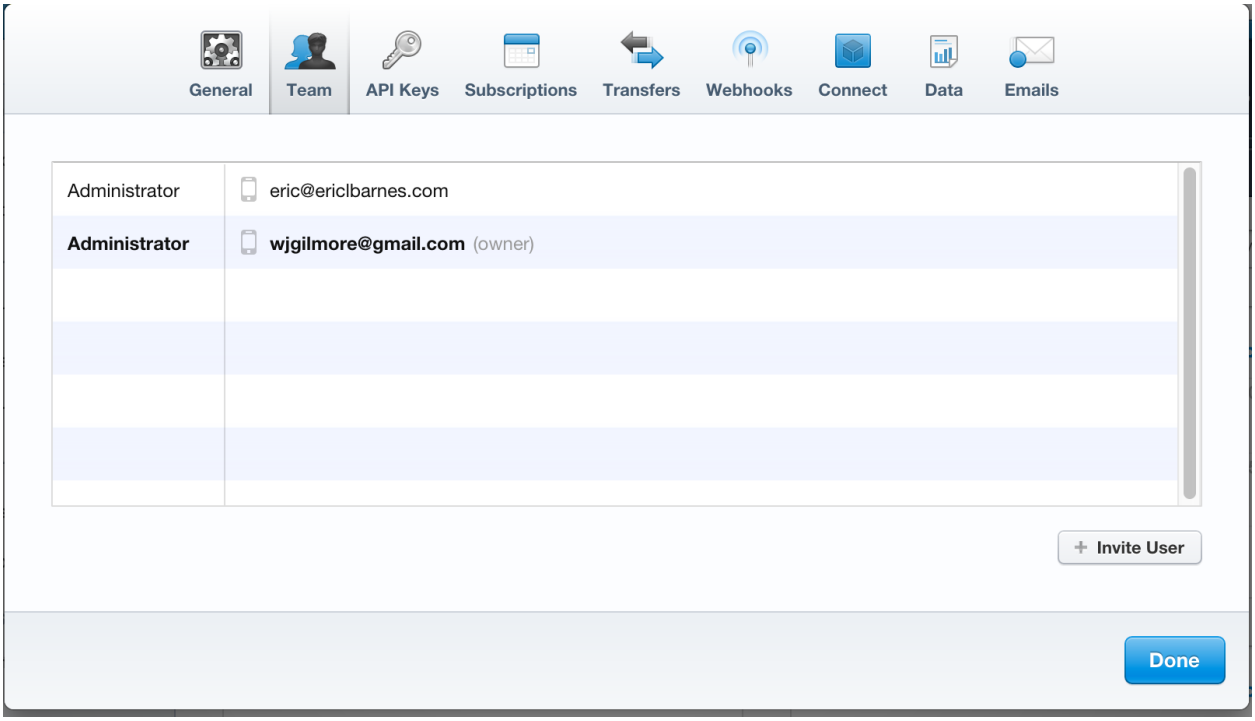
Updating Key Public Information

Let's have a look at the relevant settings:

- **Email** and **Password**: Use these settings to change your Stripe account e-mail address and password, respectively.
- **Account name**: This setting identifies your account within the Stripe dashboard. It's only relevant when you link your e-mail address to multiple Stripe accounts, such as would be the case if you were identified as a team member on multiple accounts.

- **Country**: Stripe should correctly guess your country-of-residence, however if it is incorrect (or if you're building a Stripe solution for a client residing in another country) you can change the company's country of origin here. This setting has an impact on the localization of your account, including currency, date formatting, and grammar.
- **Timezone**: This setting affects any timestamps used in reporting transactions.
- **Decline**: You might choose to apply additional credit card processing safeguards to confirm a customer's identity, including requiring the customer to identify the CVC (card verification code) and zip code associated with the credit card. If so you would enable those options here.

Adding Team Members Many e-commerce projects involve multiple developers, several of whom might benefit from having access to the Stripe dashboard. Additionally, at some point during the development process you'll need to acquaint the client's accounting and customer service teams with the dashboard so they're competently able to issue refunds and perform other customer-facing tasks. You can add team members using the Account Settings' Team tab (see below screenshot).



Managing Team Members

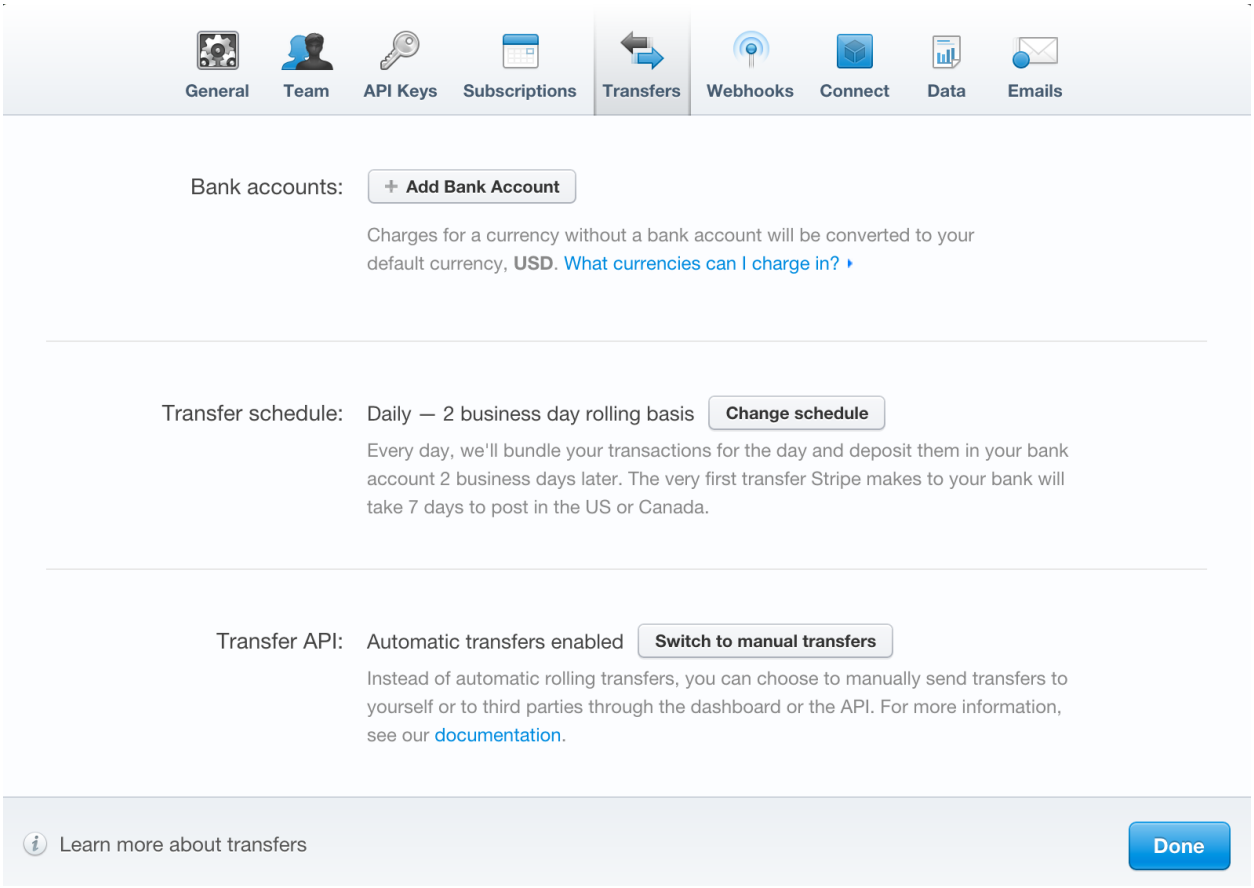
To do so, navigate to the Team tab and click the `Invite User` button. From there you'll be prompted to provide the team member's e-mail address and designate an access level:

- **Administrator**: An administrator will have the same access privileges as the account creator, meaning this user's actions carry significant weight. Therefore you should be careful to issue administrator access only to those users having an unambiguous need for this privilege level.

- **Read & Write**: Users granted read and write access are able to view *and modify* the account's customer- and payment-related data. You might grant this access level to a company employee tasked with interacting with customers and issuing refunds.
- **Read Only**: Users granted read-only access are only able to review customer and payment-related data. You might grant this access level to Mr. McDew or anybody else with an interest in monitoring the account.

Any newly invited team member will receive an e-mail and a confirmation link. Once the team member clicks on the link they'll be officially added to the account.

Adding Your Bank Account Stripe will periodically transfer any monies earned via their payment service to a designated bank account. You'll use the Account Settings' `Transfers` tab to specify the destination account, and optionally customize the transfer schedule. To add a bank account navigate to the `Transfers` tab and click the `Add Bank Account` button. You'll be prompted to provide the bank account's routing and account numbers (see below figure).



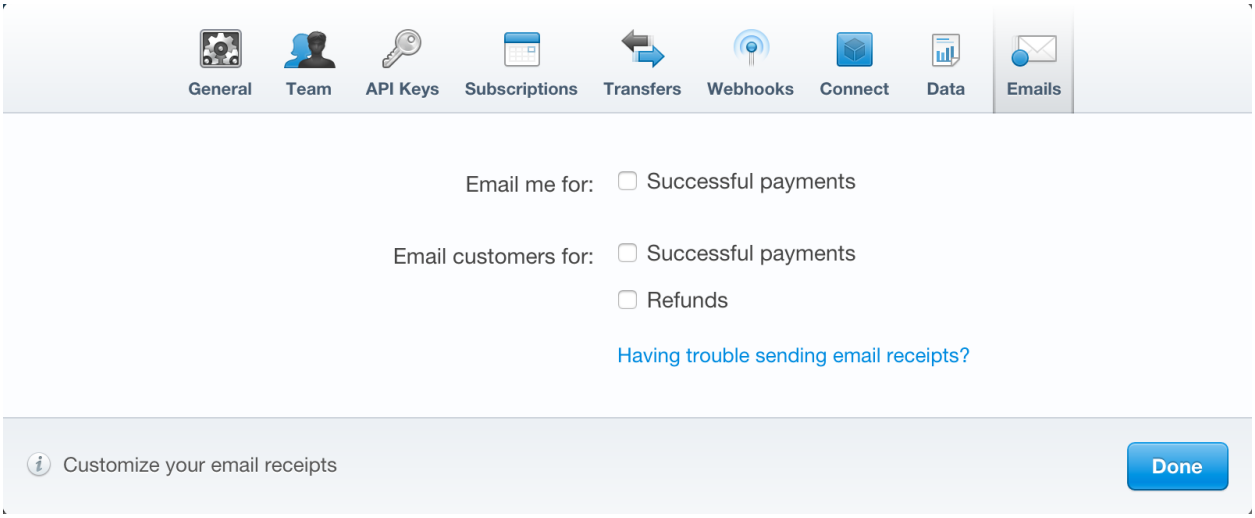
Managing Your Bank Account

In this section you can also optionally adjust the default transfer schedule. Stripe will by default transfer funds to your account two days after the transaction date, the lone caveat being the very

first transaction which will take seven days to transfer for fraud prevention reasons. You can adjust this setting to instead transfer weekly (one week after the transaction) or monthly (one month after the transaction).

Keep in mind you're not required to add a bank account in order to begin developing the application. You'll be able to use Stripe's Test environment without configuring any bank-related information, therefore there's no need to track down this information at this time.

Configuring E-mail Notifications As you'll soon learn Stripe offers a very convenient feature known as a webhook which allows you programmatically capture real-time notifications pertaining to events such as sales, Stripe additionally offers a simple e-mail-based notification alternative in which you can be sent an e-mail every time a successful payment is made. Additionally, you can choose to send customers e-mails pertaining to successful payments and refunds. If you'd like to enable any of these notification e-mails, navigate to the Account Settings' `Emails` tab (see below screenshot).



Managing Notification E-mails

Introducing Cashier

Cashier[40] is a package created and maintained by Laravel creator Taylor Otwell that greatly simplifies Stripe integration into your Laravel application. Originally created to help developers manage subscriptions, a recent update added the ability to perform one-time charges (charge a customer's credit card a single time for reasons of purchasing an electronic or physical product). This means you can now use Cashier to sell products as easily as services. We'll spend much of the remainder of this chapter integrating Cashier into the website and creating a product catalog which can be autonomously managed by Mr. McDew and his staff. Let's begin by installing and configuring Cashier.

[40] https://github.com/laravel/cashier

Installing and Configuring Cashier

Cashier is distributed as a Composer package, meaning to install Cashier all you need to do is add the following package to `composer.json`:

```
"require": {
    ...
    "laravel/cashier": "~5.0"
},
```

The Cashier 5.0+ package presumes you're using a version of the Stripe's API newer than the 2015-02-18 release, and also depends upon a more recent version of the Stripe Composer package. If you've just created a Stripe account and this statement is meaningless to you, then use the above package declaration. If you've been using the Stripe API for other projects and you'd like to continue using an API version older than the 2015-02-18 release, then you'll want to use the following declaration:

```
"require": {
    ...
    "laravel/cashier": "~5.0"
},
```

Save the changes and run `composer update` to install the package:

```
$ composer update
Loading composer repositories with package information
Updating dependencies (including require-dev)
...
- Installing laravel/cashier (v5.0.9)
  Downloading: 100%
```

Once installed, you'll next need to add the `CashierServiceProvider` to the `config/app.php` file's `providers` array:

```
'providers' => [
    ...
    'Laravel\Cashier\CashierServiceProvider',
],
```

After saving these changes we'll need to update the `User` model to support Stripe. We'll do that next!

Updating the User Model

Next you'll want to update the database to include some Cashier-specific fields, as well as your `User` model to include the `Billable` trait. Begin by executing the `cashier:table` Artisan command which is automatically made available once Cashier is installed:

```
$ php artisan cashier:table users
Migration created successfully!
```

After executing the migration (`php artisan migrate`), the `users` table will be updated to include seven new columns.

```
mysql> describe users;
+----------------------+------------------+------+-----+-----+
| Field                | Type             | Null | Key | ... |
+----------------------+------------------+------+-----+-----+
| id                   | int(10) unsigned | NO   | PRI | ... |
| name                 | varchar(255)     | NO   |     | ... |
| email                | varchar(255)     | NO   | UNI | ... |
| ...                  | ...              | ...  |     | ... |
| stripe_active        | tinyint(4)       | NO   |     | ... |
| stripe_id            | varchar(255)     | YES  |     | ... |
| stripe_subscription  | varchar(255)     | YES  |     | ... |
| stripe_plan          | varchar(100)     | YES  |     | ... |
| last_four            | varchar(4)       | YES  |     | ... |
| trial_ends_at        | timestamp        | YES  |     | ... |
| subscription_ends_at | timestamp        | YES  |     | ... |
+----------------------+------------------+------+-----+-----+
```

Frankly these seven newly added fields won't be relevant until we discuss subscription-based billing in Chapter 5, however in any case it makes sense to add them now since we've already configured other aspects of Cashier in the earlier section. Open `app/User.php` and modify the class to implement the `BillableContract`:

```
use Laravel\Cashier\Billable;
use Laravel\Cashier\Contracts\Billable as BillableContract;

class User extends Model
    implements AuthenticatableContract, CanResetPasswordContract,
    BillableContract {

    use Authenticatable, CanResetPassword, Billable;

    ...

}
```

By adding the `Billable` interface to the `User` model you can take advantage of the various payment-related methods exposed through Cashier while working with a specific user.

Next, we'll need to update the `User` model to make it a bit easier to interact with the subscription's trial and subscription conclusion dates. Add the following property to the model:

```
1  protected $dates = ['trial_ends_at', 'subscription_ends_at'];
```

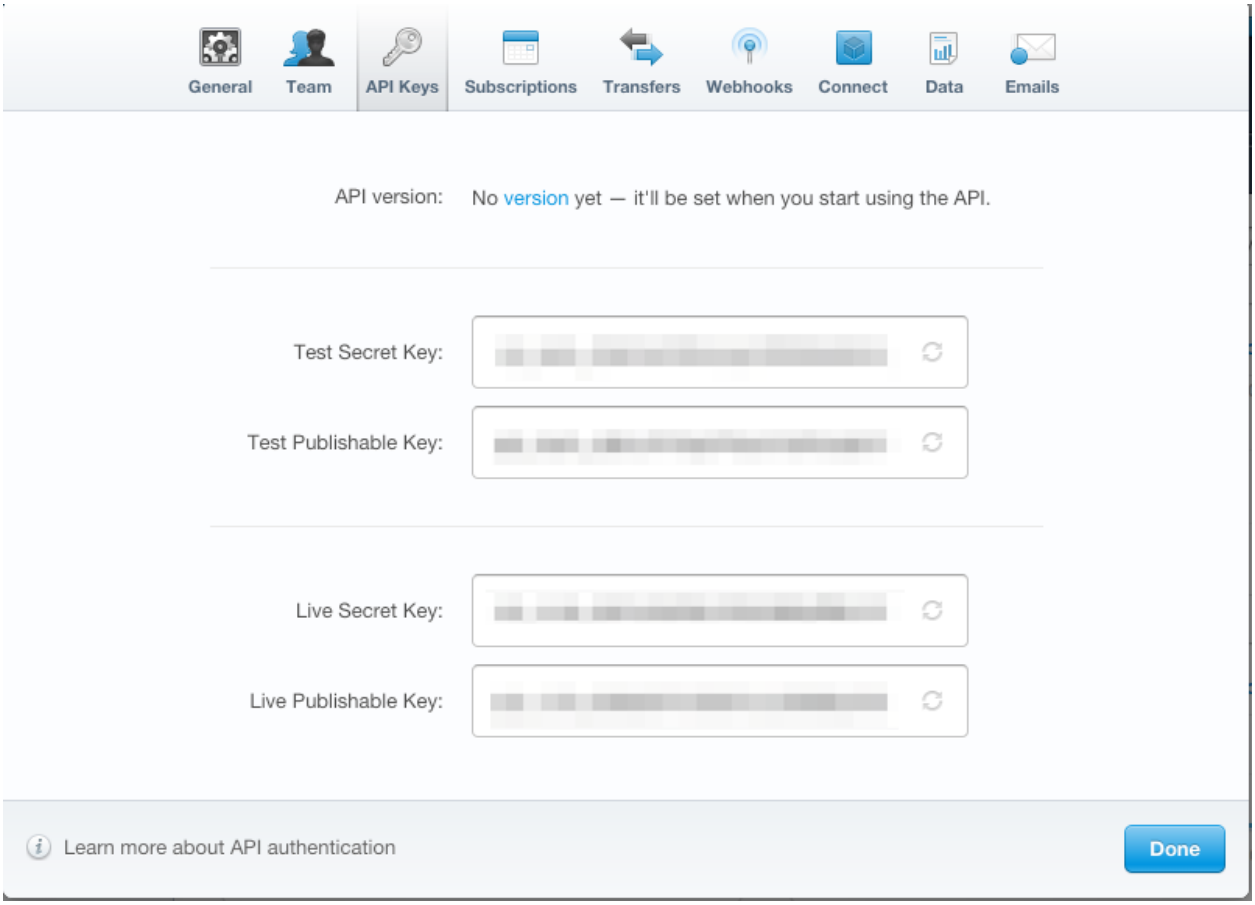
Internally this tells Laravel that we want those columns to be treated as dates which means they are automatically converted to a Carbon[41] object. If you are not familiar with Carbon, it is an extension to PHP's `DateTime` class that makes working with dates very simple.

In this or the next chapter we won't actually tie purchases to a specific user however this ability will be particularly appreciated in Chapter 5 when subscriptions are implemented.

Adding Your API Key

After saving these changes you'll need to add your Stripe API key to the `config/services.php` file. You can find your API key by returning to the Stripe dashboard and opening the `Account Settings` modal. Click on the `API Keys` tab and you'll be provided with two sets of keys (see the below screenshot), one set for development (test) and another for production (live).

[41] https://github.com/briannesbitt/Carbon



Retrieving Your Stripe API Keys

Notice how both the test and live sets consist of a secret and publishable key. There is a very importance difference between these two types of keys:

- **The Secret Key**: The Secret Key is used to perform API calls. You should *never* share this key! Should a third-party gain access to this key they could perform API calls on your behalf, potentially causing considerable damage and grief in the process.
- **The Publishable Key**: The Publishable Key is included in the forms you'll create, as it will be used to create credit card tokens. We'll talk more about exactly how you'll include this key in your forms later in the chapter.

Of course, you won't want to embed these keys directly into the `services.php` file and your forms; Instead, you'll place the keys in your `.env` file so the appropriate environment-specific key can be used. So open up your development `.env` file now and add the following two keys:

```
STRIPE_API_SECRET=sk_test_KEY_GOES_HERE
STRIPE_API_PUBLIC=pk_test_KEY_GOES_HERE
```

Then open up `config/services.php` and modify the `stripe` array to look like this:

```
'stripe' => [
    'model'  => 'User',
    'secret' => env('STRIPE_API_SECRET'),
],
```

With the changes to `services.php` in place it's time to begin building the catalog!

Creating the Product Catalog

Next let's turn our attention towards creating the product catalog. Mr. McDew only wants to sell a few products to get started, which is ideal because we can devote the majority of time to building a convenient product administration interface and ensuring proper Stripe integration. Let's kick things off by creating a new model named `Product` which will manage the catalog products.

Creating the Product Model

We'll use a model named `Product` to manage the products available for sale through the website. Each product will include a name, SKU, description, price, and picture. Additionally, a Boolean flag will determine whether the product is physical or downloadable. Because only a few products will be sold we won't create categories however as a programming exercise consider extending the application to associate each product with a category using a belongsTo/hasMany relationship! Let's create the `Product` model:

```
$ php artisan make:model Product -m
Model created successfully.
Created Migration: 2015_03_31_013030_create_products_table
```

Next, open the newly created migration and modify the `up()` method to look like this:

```
public function up()
{
    Schema::create('products', function(Blueprint $table)
    {
        $table->increments('id');
        $table->string('sku')->unique();
        $table->string('name');
        $table->text('description');
        $table->decimal('price', 6, 2);
        $table->boolean('is_downloadable')->default(false);
        $table->timestamps();
    });
}
```

The purpose of each field is likely obvious, except for `is_downloadable`. This field is used to differentiate between electronic (e.g. a downloadable lawncare guide) and physical products (e.g. a lawnmower). If `is_downloadable` is `true`, then the product is electronic and a slightly different post-purchase process will be taken than if it were physical. We'll talk more about this process in the next chapter.

Creating the Catalog Controller

Next let's create a controller for displaying the products to site visitors. We'll keep this simple, providing just a list and detail view. Even so let's save some typing by creating a resourceful controller and then removing all of the actions except `index` and `show`:

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

Now open the newly generated `ProductController.php` and delete the `create`, `store`, `edit`, `update` and `destroy` methods, and then update the `index` and `show` methods so the resulting controller looks like this:

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

use App\Http\Requests;
use App\Http\Controllers\Controller;
use App\Product;
use Illuminate\Http\Request;

class ProductController extends Controller {

    public function index()
    {
        $products = Product::orderBy('name', 'asc')->get();
        return view('product.index', compact('products'));
    }

    public function show($id)
    {
        $product = Product::findOrFail($id);
        return view('product.show', compact('product'));
    }

}
```

Finally, open `routes.php` and add the resourceful route, being sure to limit the available actions to `index` and `show`:

```
Route::resource('products', 'ProductController',
    array('only' => 'index, show'));
```

This will result in the product listing being presented via the URI `/products` and a product detail page via a URL such as `/products/2`. If you'd rather use a URI such as `/catalog` then you can just change the route declaration to look like this:

```
Route::resource('catalog', 'ProductController',
    array('only' => 'index, show'));
```

Although for the purposes of this exercise we're just passing product primary keys via the URL in order to properly identify a product in the detail view, for real-world purposes we recommend using a more friendly URL syntax for both UX and SEO reasons. A package named eloquent-sluggable[42] greatly reduces the amount of work required to incorporate user-friendly URLs into your Laravel application.

[42] https://github.com/cviebrock/eloquent-sluggable

Creating an Administration Console

Mr. McDew insists upon easily updating the product names, prices and descriptions using a web interface, which certainly seems like a reasonable enough request. One of the easiest ways to integrate this feature into the site is via a restricted administration console. This console would be accessible only by those users deemed to be an administrator. Laravel's middleware capabilities are perfectly suited for this sort of feature. Using middleware, you can set a Boolean flag in the `users` table called `is_admin`, and then confirm the flag is enabled prior to granting access to any of the administrative controllers. Let's begin by updating the `users` table.

Adding an Administrator Flag to the users Table

A Boolean flag can be set within the `users` table which will determine whether the user is identified as an administrator. Let's add that field now:

```
$ php artisan make:migration add_is_admin_to_user_table --table=users
Created Migration: 2015_03_31_171009_add_is_admin_to_user_table
```

Open the newly created migration file and modify the `up()` and `down()` methods to look like this:

```
public function up()
{
    Schema::table('users', function(Blueprint $table)
    {
        $table->boolean('is_admin')->default(false);
    });
}

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

Save the changes and run the migration to add the column. Note that because the `is_admin` column will default to `false`, all existing users' `is_admin` column will automatically be disabled, meaning the soon-to-be-created middleware will treat them as non-administrators and decline access to the restricted administration console. Therefore in order to identify certain users as administrators you'll need to login to your database and update the appropriate records, setting `is_admin` to `true`.

Creating the Administration Middleware

With the administrator flag added to the `users` table, let's next create the middleware which will be used by the restricted administration controllers to determine whether the requesting user should be granted access:

```
$ php artisan make:middleware AdminAuthentication
Middleware created successfully.
```

You'll find the newly generated middleware skeleton inside `app/Http/Middleware/AdminAuthentication.php`. Open it and make the following changes, beginning with adding the following lines to the top of the file:

```
use Illuminate\Contracts\Auth\Guard;
use Illuminate\Http\RedirectResponse;
```

Next, add the following protected attribute and constructor to the `AdminAuthentication` class:

```
protected $auth;

public function __construct(Guard $auth)
{
    $this->auth = $auth;
}
```

Finally, modify the `handle()` method to look like this:

```
public function handle($request, Closure $next)
{
    if ($this->auth->check())
    {
        if ($this->auth->user()->is_admin == true)
        {
            return $next($request);
        }
    }

    return new RedirectResponse(url('/'));

}
```

The `handle` method automatically executes whenever the middleware is triggered. It will first determine whether the requesting users is signed in, and if so, will next check if the user's `is_admin` table column is set to true. If so, the request is allowed to proceed (`return $next($request)`). If the user isn't signed in or is not identified as an administrator, he will be redirected to the home page (`return new RedirectResponse(url('/'))`).

After saving the changes to `AdminAuthentication.php`, you'll need to register the middleware with your application. This is done by adding a reference to the `app/Http/Kernel.php`'s `$routeMiddleware` array:

```
1  protected $routeMiddleware = [
2      ...
3      'admin' => 'App\Http\Middleware\AdminAuthentication',
4  ];
```

After saving those changes it's time to create the administration controllers and associated views. After doing so, we'll return to the administration middleware in order to restrict access to the administration console.

Creating an Administration Controller

With the `users` table updated and the administration middleware in place, it's time to create the restricted administration controllers and views. Begin by creating an administration controller we'll use for managing the products:

```
1  $ php artisan make:controller Admin/ProductController
```

Note the `Admin/` prefix. Adding this prefix will result in a new directory named `Admin` being created inside `app/Http/Controllers`, where the `ProductController.php` file will be placed. Among other advantages this allows us to create controllers named identically to those found in the `app/Http/Controllers` directory, meaning you won't have to create any awkwardly-named controllers (e.g. `AdminproductController`). Once created you can create a prefixed route grouping that looks like the following:

```
1  Route::group(['prefix' => 'admin', 'namespace' => 'Admin'], function()
2  {
3      Route::resource('products', 'ProductController');
4  });
```

The `prefix` setting allows the administration controllers to be accessed using a URI prefixed with `/admin/`, meaning for instance the product administration controller's `index` action will be accessed using the URI `/admin/products`.

With the administrative area configured, we suggest creating the typical CRUD interfaces for managing your products. Because such features are fairly standard we won't belabor the process here, however of course should you have any questions be sure to have a look at how these controllers and associated views are implemented in the project source code.

Restricting Access to the Administration Console

After you have thoroughly tested the catalog administrator, it's time to restrict access to the administration area. Because we've already created the administration middleware responsible for restricting access, all that remains is to apply it to the administration route grouping. Return to your `routes.php` file and add `'middleware' => 'admin'` to the `Route::group` array:

```
1  Route::group([
2      'prefix' => 'admin',
3      'namespace' => 'Admin',
4      'middleware' => 'admin'
5  ], function() {
6      Route::resource('products', 'ProductController');
7  });
```

Once in place, only those authenticated users having `is_admin` set to `true` will be able to access the administration console!

Summary

In this chapter we integrated Laravel Cashier, and created a restricted administration console for managing products. With these pieces in place, we'll move on to the second step of this phase which involves integrating the Stripe checkout interface, and extending the administration console to track orders.

Chapter 4. Selling Electronic and Physical Products

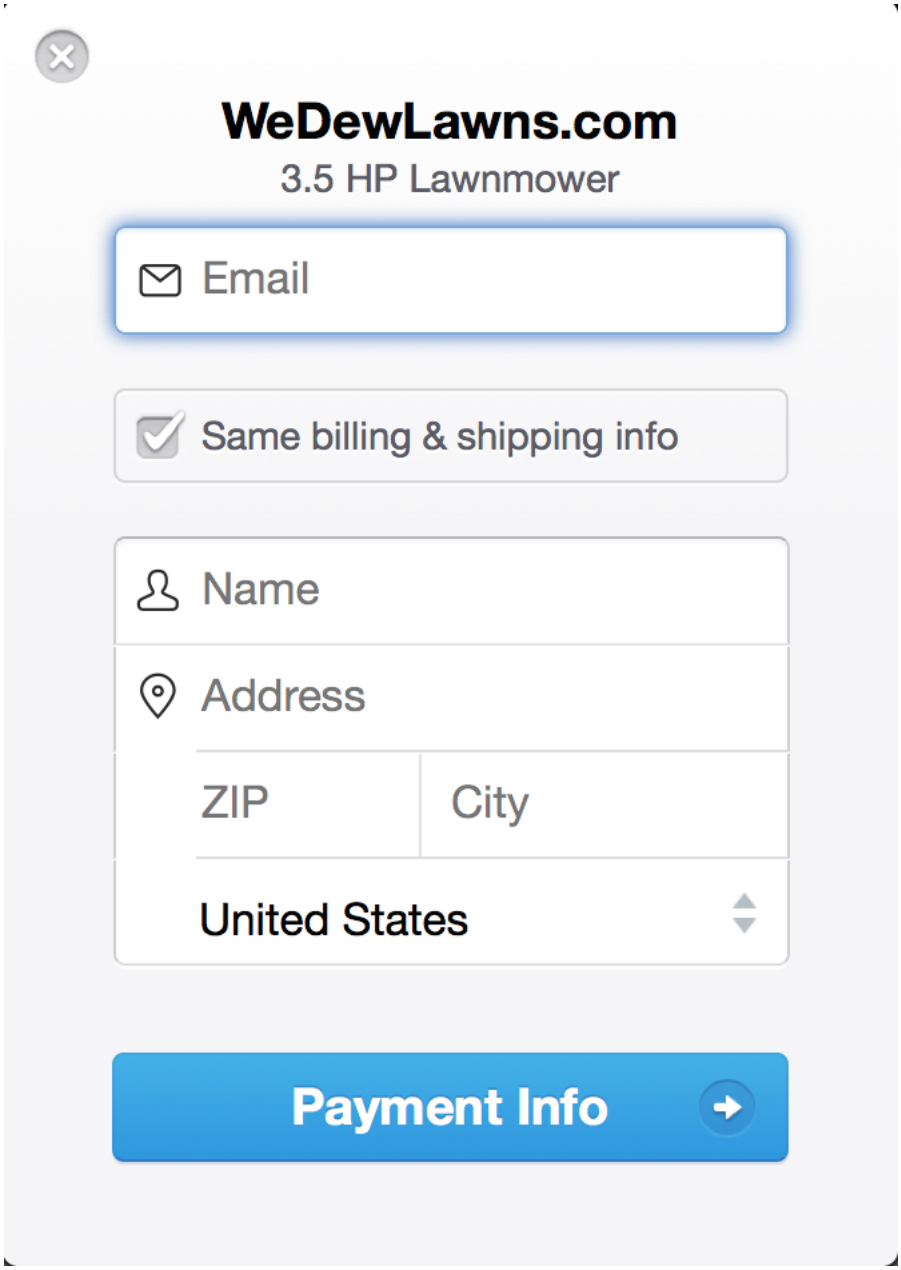
Electronic and physical products are sold in vast quantities via online stores each and every day. Yet each product type presents a unique set of challenges in terms of online sales, so in this chapter we'll work through examples demonstrating how to integrate each product type into your online catalog and sell it to eager customers using Cashier and Stripe.

Purchasing Products Using Stripe

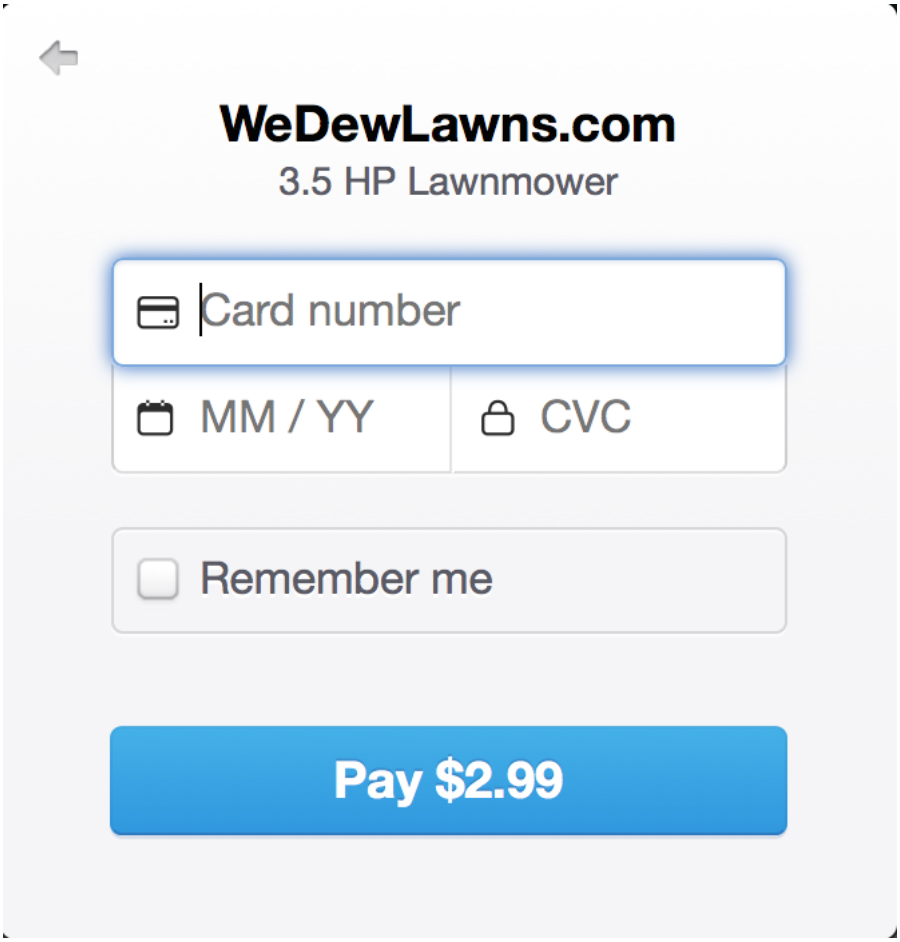
In the last chapter you integrated Laravel Cashier into the site, however Cashier is only one part of the puzzle. Among other things, Cashier facilitates the Stripe API call which will charge the customer's credit card. However prior to charging the card you'll want to validate the credit card information (number, expiration date, security code). Additionally, you do *not* want the customer's credit card information to ever touch your server, as handling that information in any fashion would require a whole host of PCI compliance[43] measures to be implemented.

Stripe offers a very convenient workaround to the PCI compliance difficulties by presenting the checkout form in a secure modal (see below screenshots).

[43] http://en.wikipedia.org/wiki/Payment_Card_Industry_Data_Security_Standard



The Stripe Checkout Modal (Step #1)



The Stripe Checkout Modal (Step #2)

The credit card information is sent directly to the Stripe servers, and a unique *token* is returned. You'll then pass this token along with other purchase-related details to the Stripe API via Cashier. Once the Stripe service receives the information, it will use the token to obtain the customer's credit card and complete the purchase process.

As you'll soon see, integrating the Stripe checkout modal is incredibly easy. We'll do that next, followed by using Cashier to complete the transaction via the Stripe API.

Integrating the Stripe Purchase Button

Head over to http://wedewlawns.com/products[44] and you'll find a list of products sold through the We Dew Lawns website. Feel free to have a look at the source code and even complete a transaction (using *fake* credit card information). The site is of course not real and when you click on a product's `Buy` button you'll see a yellow `Test Mode` label located at the top right of the browser screen. Stripe automatically adds this label when the service is running in test mode, meaning no credit cards are actually processed.

[44] http://wedewlawns.com/products

Each Buy button is actually generated by a bit of JavaScript hosted by Stripe. Here's a snippet of the code used on the website to create these buttons:

```
{!! Form::open(array('url' => '/checkout')) !!}
{!! Form::hidden('product_id', $product->id) !!}
<script
src="https://checkout.stripe.com/checkout.js" class="stripe-button"
data-key="{{env('STRIPE_API_PUBLIC')}}"
data-name="WeDewLawns.com"
data-billingAddress=true
data-shippingAddress=true
data-label="Buy ${{ $product->price }}"
data-description="{{ $product->name }}"
data-amount="{{ $product->priceToCents() }}">
</script>
{!! Form::close() !!}
```

As you can see, a `<script>` element references `https://checkout.stripe.com/checkout.js`, and identifies eight additional attributes (`class`, `data-key`, `data-name`, etc.) which are used to tweak the behavior and appearance of the Buy button and checkout modal. Let's review the purpose of each attribute:

- `class`: Setting the `class` attribute to `stripe-button` stylizes the Buy button using the default blue theming. It is possible to customize the button styling although doing so is out of the scope of this book; see the Stripe documentation[45] for more information about customization options.
- `data-key`: The `data-key` attribute is used to pass your publishable API key to Stripe. Refer back to the section "Adding Your API Key" in Chapter 3 for more information about making this key available to your application.
- `data-name`: The `data-name` attribute identifies the name of your website or company. If you refer back to the Stripe checkout modal screenshot you'll see this value is used at the top of the modal.
- `data-billingAddress`: When `data-billingAddress` is set to `true`, the checkout modal will require the user to supply his billing address.
- `data-shippingAddress`: When `data-shippingAddress` is set to `true`, the checkout modal will require the user to supply his shipping address. Logically this is useful when the customer is purchasing a physical product.
- `data-label`: The `data-label` field contains the text used in the Buy button.
- `data-description`: The `data-description` field contains the text found below the company or website name in the checkout modal. Referring back to the earlier screenshot, this is set to "3.5 HP Lawnmower".

[45] https://stripe.com/docs/checkout#integration-custom

- `data-amount`: The `data-amount` field contains the amount of money the user's credit card will be charged. This value will be displayed in the payment submission button (`Pay $4.99` in the second step of the payment process as depicted in the early screenshot). It is very important for you to understand this is *only* used for purposes of displaying the price. You will subsequently use Cashier to identify exactly how much the customer's card will be charged. Also, notice how we use a `Product` model method named `priceToCents()` to identify this value. This is because Stripe requires the value to be provided in pennies, meaning `$4.99` will need to be identified as `499`.

The `Product` model `priceToCents` method used to convert the product price into a penny-based representation is very simple:

```
public function priceToCents()
{
    return $this->price * 100;
}
```

The data found in the `<script>` element will be submitted via Ajax, and if successful Stripe will embed several hidden elements into the form, including the aforementioned Stripe token, and various other important bits of information such as the customer's billing and shipping addresses (if you configured the form to require this information). For instance, given the above `Buy` button `<script>` element configuration, Stripe will embed the following fields into the form following a successful submission:

```
<input type="hidden" name="stripeToken" value="tok_15rtKuBJbKzGetVOPKP">
<input type="hidden" name="stripeTokenType" value="card">
<input type="hidden" name="stripeEmail" value="wjgilmore@example.com">
<input type="hidden" name="stripeBillingName" value="Jason Gilmore">
<input type="hidden" name="stripeBillingAddressLine1" value="12 Jump Ct.">
<input type="hidden" name="stripeBillingAddressZip" value="43016">
<input type="hidden" name="stripeBillingAddressState" value="OH">
<input type="hidden" name="stripeBillingAddressCity" value="Dublin">
<input type="hidden" name="stripeBillingAddressCountry"
  value="United States">
<input type="hidden" name="stripeBillingAddressCountryCode" value="US">
<input type="hidden" name="stripeShippingName" value="Jason Gilmore">
<input type="hidden" name="stripeShippingAddressLine1" value="12 Jump Ct.">
<input type="hidden" name="stripeShippingAddressZip" value="43016">
<input type="hidden" name="stripeShippingAddressState" value="OH">
<input type="hidden" name="stripeShippingAddressCity" value="Dublin">
<input type="hidden" name="stripeShippingAddressCountry"
  value="United States">
<input type="hidden" name="stripeShippingAddressCountryCode" value="US">
```

By injecting this additional information into the form, you'll be able to access it in order to complete the transaction and optionally additionally record the order details to your local database. The information will be sent to whatever endpoint you've defined in the form's action attribute, which in the case of the above example is set to the `Checkout` controller. We'll create this controller in just a bit, however first let's turn our attention to creating a facility for recording order information.

Managing the Order Information

In a moment we'll create the `Checkout` controller referenced in the purchase form, however first let's create the model and corresponding table used to locally record the order information. This information will be stored in a table named `orders` and managed via a corresponding model named `Order`, so let's create those now:

```
$ php artisan make:model Order -m
Model created successfully.
Created Migration: 2015_04_15_165132_create_orders_table
```

Next, open the newly created migration file and update the `up()` method to look like this:

```
public function up()
{
    Schema::create('orders', function(Blueprint $table)
    {
        $table->increments('id');
        $table->integer('product_id')->unsigned();
        $table->foreign('product_id')->references('id')->on('products');
        $table->string('email');
        $table->string('billing_name');
        $table->string('billing_address');
        $table->string('billing_city');
        $table->string('billing_state');
        $table->string('billing_zip');
        $table->string('billing_country');
        $table->string('shipping_name')->nullable();
        $table->string('shipping_address')->nullable();
        $table->string('shipping_city')->nullable();
        $table->string('shipping_state')->nullable();
        $table->string('shipping_zip')->nullable();
        $table->string('shipping_country')->nullable();
        $table->string('onetimeurl')->nullable();
        $table->timestamps();
    });
}
```

The purpose of each field defined in this migration should be apparent, except for perhaps `onetimeurl`. This field is used in conjunction with the purchase of an electronic (downloadable) product. When the purchase is completed, a unique URL intended for "one time" use will be generated and stored in this field. The customer is then sent an e-mail containing this URL. When the customer clicks on the link the download process will be initiated. We'll talk more about this process in the later section, "Downloading Electronic Products".

After saving the changes run `php artisan migrate` to create the `orders` table.

Most of the columns declared in the migration are used to manage billing and shipping information, however note we're also referencing the purchased product's primary key:

```
$table->integer('product_id')->unsigned();
$table->foreign('product_id')->references('id')->on('products');
```

This means we're defining a belongsTo relationship in which each order belongs to a product. Conversely, each product can have many orders. Therefore you'll next need to open the `Product` model and add the following association declaration:

```
  public function orders()
  {
    return $this->hasMany('App\Order');
  }
```

Save the changes and then open the `Order` model, adding the following association declaration:

```
  public function product()
  {
    return $this->belongsTo('App\Product');
  }
```

Adding this relationship between the `Product` and `Order` models is certainly useful, however it raises an interesting dilemma. What if one of the products are discontinued and therefore subject to removal from the store? Deleting a product associated with past orders means the `orders` table's `product_id` would be orphaned. Although all past order-related information is also available for review using the Stripe administration console, corrupting the data found in the WeDewLawns.com administration console would be a major inconvenience so let's modify the `Product` model to ensure products are *soft deleted* rather than deleted outright.

When you soft delete a product, a `deleted_at` timestamp column found in the `products` table will be updated. Laravel will automatically look for this column and any records having a set timestamp will be filtered from selection results. Because soft deleted products still exist in the table though, any relations to orders will not be broken. To enable this feature you'll need to modify the `Product` model to use the `SoftDeletes`:

```
use Illuminate\Database\Eloquent\SoftDeletes;

class Product extends Model {

    use SoftDeletes;

    protected $dates = ['deleted_at'];

    ...

}
```

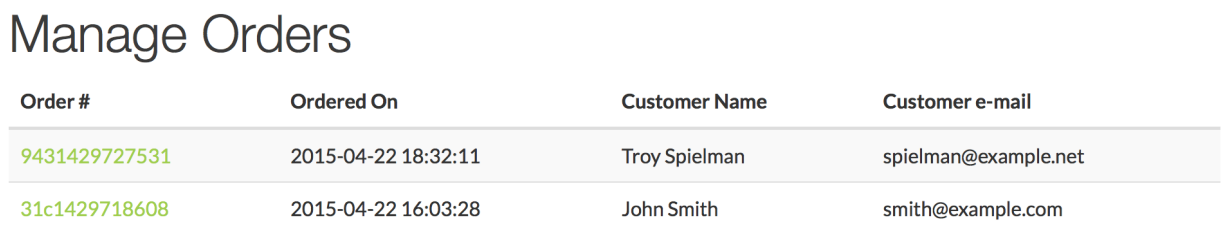
Next, create a migration which will add the `deleted_at` column to the `products` table:

```
$ php artisan make:migration add_deleted_at_to_products_table --table=products
Created Migration: 2015_04_16_174155_add_deleted_at_to_products_table
```

Open the newly created migration file and update the `up()` method to look like this:

```
public function up()
{
    Schema::table('products', function(Blueprint $table)
    {
        $table->softDeletes();
    });
}
```

Run the migration to add the column and you're done! With these updates in place you can go about creating an order review interface similar to the one found in the administration console (see below screenshot).



The WeDewLawns.com Order Review Console

See the Laravel Documentation[46] for more information about other soft deletion features.

[46] http://laravel.com/docs/master/eloquent#soft-deleting

Completing the Checkout Process

When the hidden fields found in the purchase form are submitted to the `/checkout` endpoint, the associated `Checkout` controller will have access to the form data just like any other typical form submission. That form data will be made available via the `$request` object. For instance if you dump the `$request` object contents (using `dd()` for example) you'll be presented with the following array:

```
array:18 [
  "_token" => "ISH5o3abrUxleE6y4MIvOrHjoQVQFZ7LirzaKw9a"
  "stripeToken" => "tok_15pOyvBJbKzGteVOZH3g10ob"
  "stripeTokenType" => "card"
  "stripeEmail" => "wjgilmore@example.com"
  "stripeBillingName" => "Jason Gilmore"
  "stripeBillingAddressLine1" => "1234 Jump Street"
  "stripeBillingAddressZip" => "43016"
  "stripeBillingAddressState" => "OH"
  "stripeBillingAddressCity" => "Dublin"
  "stripeBillingAddressCountry" => "United States"
  "stripeBillingAddressCountryCode" => "US"
  "stripeShippingName" => "Jason Gilmore"
  "stripeShippingAddressLine1" => "1234 Jump Street"
  "stripeShippingAddressZip" => "43016"
  "stripeShippingAddressState" => "OH"
  "stripeShippingAddressCity" => "Dublin"
  "stripeShippingAddressCountry" => "United States"
  "stripeShippingAddressCountryCode" => "US"
]
```

This means that you can easily retrieve this form data using the `$request` object's `input()` method. For instance to retrieve the `stripeToken` field value you'll use the following statement:

```
$stripeToken = $request->input('stripeToken');
```

Of course, we'll want to retrieve much more than just the token, because it's in the `Checkout` controller that you'll use Cashier to complete the transaction via the Stripe API and perform any other post-purchase processing such as recording the purchase locally for later fulfillment. In the case of WeDewLawns.com, the `Checkout` controller's `index` action is responsible for completing three tasks:

- Using Cashier to complete the transaction via the Stripe API.

- Recording the order information in a local database table for subsequent reference by the company's order fulfillment staff.
- Redirecting the customer to a thank you page.

Here's the `Checkout` controller code in its entirety:

```
public function index(Request $request)
{

    $user = new User();

    $product = Product::find($request->input('product_id'));

    $stripeEmail = $request->input('stripeEmail');

    $stripeToken = $request->input('stripeToken');
    $stripeTokenType = $request->input('stripeTokenType');

    if ( $user->charge($product->priceToCents(),
            ['source' => $stripeToken,'receipt_email' => $stripeEmail])
        ) {

      $order = new Order();

      // Generate random order number
      $order->order_number =
        substr(md5(microtime()),rand(0,26),3) . time();

      $order->product_id = $product->id;

      $order->email = $request->input('stripeEmail');
      $order->billing_name = $request->input('stripeBillingName');
      $order->billing_address =
        $request->input('stripeBillingAddressLine1');
      $order->billing_city =
        $request->input('stripeBillingAddressCity');
      $order->billing_state =
        $request->input('stripeBillingAddressState');
      $order->billing_zip = $request->input('stripeBillingAddressZip');
      $order->billing_country =
        $request->input('stripeBillingAddressCountry');

```

```
37        $order->shipping_name = $request->input('stripeShippingName');
38        $order->shipping_address =
39          $request->input('stripeShippingAddressLine1');
40        $order->shipping_city = $request->input('stripeShippingAddressCity');
41        $order->shipping_state =
42          $request->input('stripeShippingAddressState');
43        $order->shipping_zip = $request->input('stripeShippingAddressZip');
44        $order->shipping_country =
45          $request->input('stripeShippingAddressCountry');
46
47        $order->save();
48
49        if ($order->product->is_downloadable) {
50
51          $order->onetimeurl = md5(time() . $order->email .
52            $order->order_number);
53          $order->save();
54
55          $data = ['order' => $order];
56
57              \Mail::send('emails.download',
58                  $data,
59                  function($message) use ($data)
60              {
61                  $message->from(env('MAIL_FROM'));
62                  $message->to($data['order']->email,
63                    $data['order']->billing_name);
64                  $message->subject('WeDewLawns.com Download Instructions');
65              });
66
67        }
68
69      } else {
70
71        return \Redirect::route('products.show', [$product->id]
72          ->with('message', 'There was a problem completing your order.');
73
74      }
75
76      return \Redirect::route('checkout.thankyou')
77        ->with('message', 'Thank you for your order.');
78
```

```
79 }
```

This is a pretty straightforward bit of logic, first retrieving the e-mail, Stripe token, and Stripe token type from the request parameters and then using Cashier's `charge()` method to complete the transaction. Note the `charge()` method is made available through the `User` model, because of the `Billable` interface we added to the model earlier in the chapter. Once completed, the new order record is created, and the user is finally redirected to the order confirmation page.

Also, pay particular attention to what happens should the purchased product be electronic (the `is_downloadable` field is set to `true`). If it is electronic, a unique (one-time) URL is generated, and the customer is sent an e-mail containing instructions regarding downloading the electronic product. We'll talk more about this process in the section, "Downloading Electronic Products".

An order confirmation number is also randomly generated using a fairly simplistic algorithm. Logically your particular store might require these order numbers be generated using a different approach, however regardless you'll certainly want to ensure some sort of order number is generated so as to ensure the customer can provide you with this identifying piece of information should any questions arise.

Downloading Electronic Products

Following successful purchase of an electronic (downloadable) product, the customer is sent an e-mail containing download instructions. The most important part of these instructions is the *one-time URL*, which looks like this:

```
1 https://wedewlawns.com/product/download/3a0a57e2c1caae49c768f7bb1706c486
```

The lengthy string attached to the end of the URL is a unique key which will be used to retrieve the associated order and subsequently the electronic product purchased by the customer. The product's downloadable file name (e.g. `lawncare-guide.php`) is retrieved and then used to retrieve the actual file from its location on the server. Finally, Laravel's `response` helper is used to initiate the download. You'll find the action responsible for this task inside the `Product` controller's `download` action:

```
1  public function download($id) {
2
3    $order = Order::where('onetimeurl', $id)->first();
4
5    if ($order) {
6
7      $product = $order->product;
8
9      $order->onetimeurl = '';
10     $order->save();
```

```
11
12      return response()
13        ->download(storage_path().'/downloads/' . $product->download);
14
15    } else {
16      abort(401, 'Access denied');
17    }
18
19  }
```

As you can see, the onetime URL associated with the order is deleted immediately prior to triggering the download, preventing the customer from distributing the one-time URL. Of course, the merits of doing so are questionable given a customer could just as easily distribute the downloaded file (barring some sort of DRM, which is in itself of questionable practicality). Logically you'll want to spend some time mulling over the tradeoffs associated with protecting your IP and inconveniencing customers.

Integrating Shipping and Tax Information

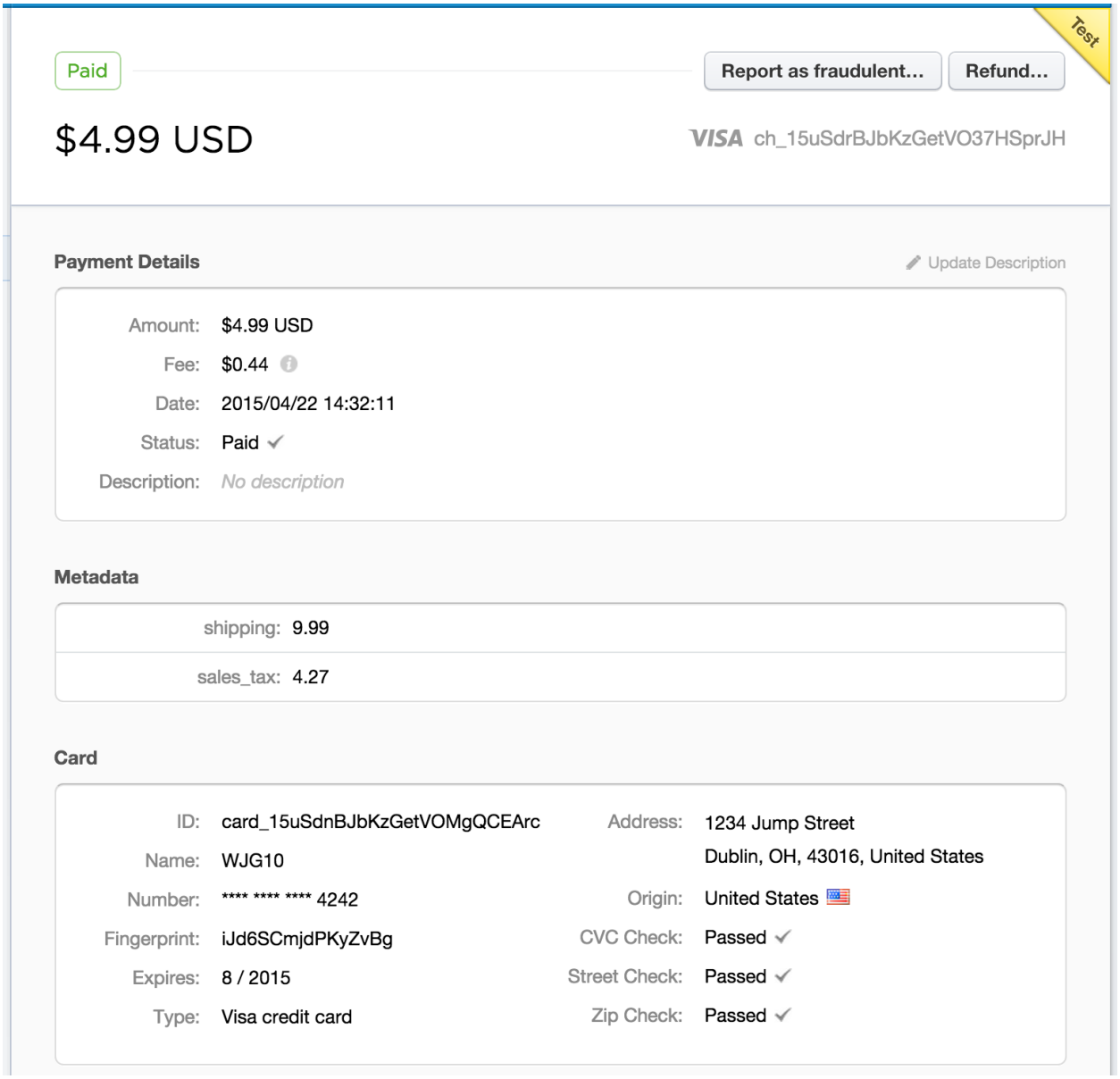
The complexities associated with integrating shipping costs and sales tax into an online store are such that an entire book could be devoted to the matter (seriously). For this reason we have decided to forgo providing any specific technical direction on these topics, as any examples will almost certainly apply to a small segment of the total readership. Instead, we thought it much more useful to discuss these matters in the context of Stripe, explaining how shipping and taxation can be recorded along with the other order information. We'll conclude by identifying a few third-party solutions you'll want to investigate when sorting out your own project's shipping and sales tax requirements.

To begin, keep in mind Stripe *does not support* calculation of sales tax shipping fees. In fact, there isn't even an obvious way for you to persist this information separately from the product price. To charge a customer shipping and/or sales tax you'll need to *add* it to the value passed into the purchase form's `data-amount` field. The Stripe team does not try to hide this deficiency and in fact devotes an entire page to the matter[47] on their support site. They do however provide a solution for at least recording this information by passing it along as *metadata*. The following snippet demonstrates how this is done using Cashier:

[47] https://support.stripe.com/questions/custom-fields-for-tax-tips-shipping-and-more

```
$user->charge($product->priceToCents(),
    [
        'source' => $stripeToken,
        'receipt_email' => $stripeEmail,
        'metadata' => ['shipping' => '9.99', 'sales_tax' => '4.27']
    ]
);
```

You're free to use any metadata keys you desire; `shipping` and `sales_tax` just happen to be the keys we chose for this example. After successfully completing the purchase you can sign into the Stripe administration console and navigate to the purchase where you'll see the purchase metadata displayed in a separate section (see below screenshot).



Viewing purchase metadata inside Stripe

Of course, you'll require a solution for calculating shipping and sales tax data in the first place. Regarding sales tax, the Stripe documentation[48] identifies the following third-party solutions as suitable sales tax management solutions:

- Avalara[49]
- BBillbo[50]

[48] https://support.stripe.com/questions/custom-fields-for-tax-tips-shipping-and-more
[49] http://www.avalara.com/
[50] https://github.com/piesync/billbo

- Octobat[51]
- Quaderno[52]
- Taxamo[53]
- TaxCloud[54]

Keep in mind this is hardly a comprehensive list, and there are plenty of other sales tax calculation solutions available; we just wanted to highlight these since they are at least implicitly endorsed by Stripe. Regarding shipping, there are dozens if not hundreds of available solutions. For instance, all three of the major shipping providers in the United States offer APIs:

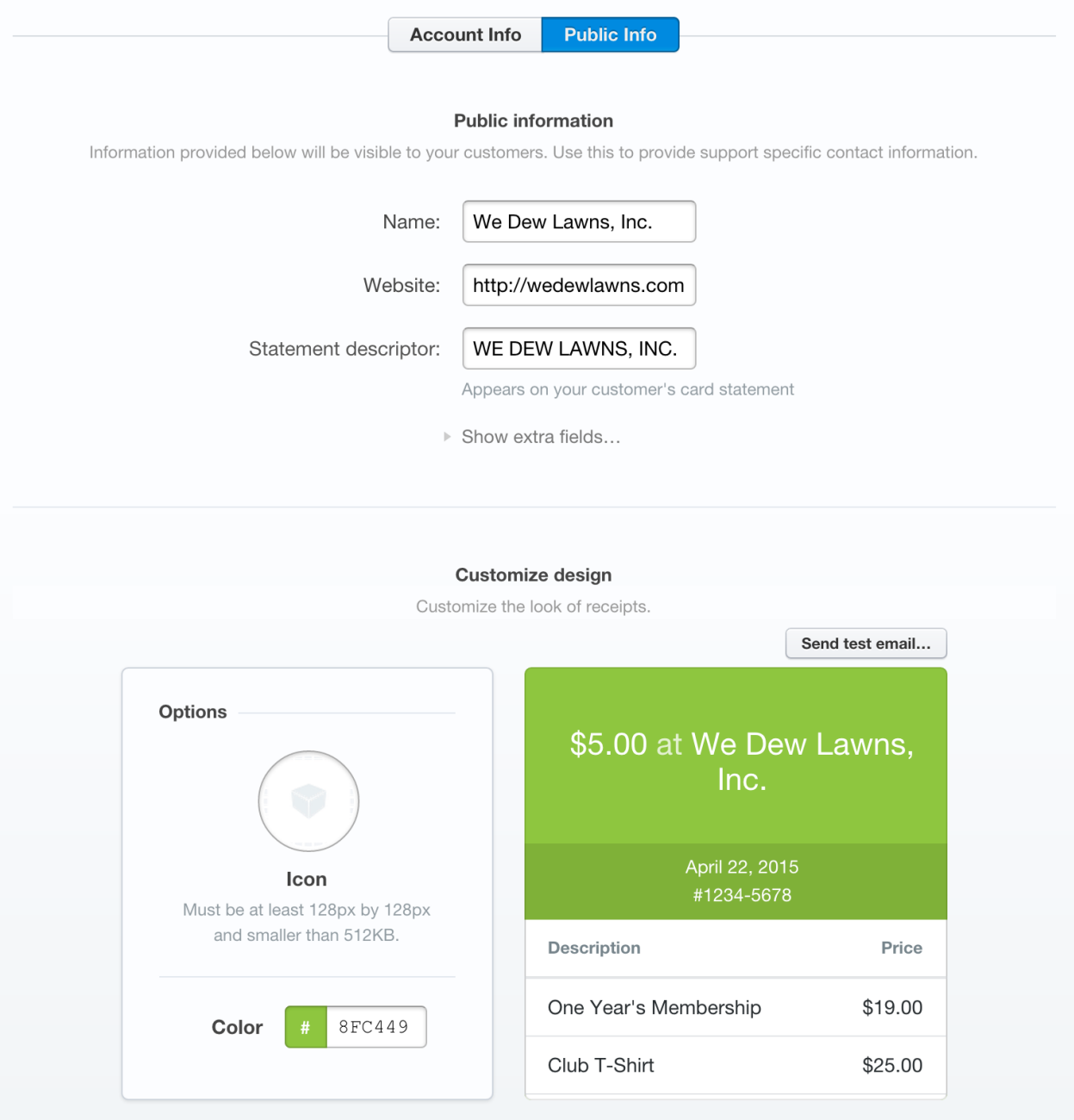
- Fedex Developer Resource Center[55]
- UPS Developer Kit[56]
- USPS Web Tools APIs[57]

Of course, you might be interested in a turnkey calculation service which offers multiple carrier options. For this we suggest turning to your favorite search engine for some research as the list is so long and varied that any we identify here would be but a drop in the bucket.

Customizing Your E-mail Receipts

If you've opted to e-mail receipts to customers (`Account Settings > Emails`), you can customize the receipt by navigating to `Account Settings` and clicking on the `Public Info` button inside the `General` tab (see below screenshot).

[51] https://www.octobat.com/
[52] https://quaderno.io/
[53] https://www.taxamo.com/
[54] https://taxcloud.net/
[55] http://www.fedex.com/us/developer/
[56] https://www.ups.com/upsdeveloperkit
[57] https://www.usps.com/business/web-tools-apis/welcome.htm



Customizing the e-mail receipt

Here you'll be able to customize the e-mail header background color, add a logo, and add a custom company icon. If you click the `Show extra fields...` link you can additionally add a support e-mail address, phone number, URL, and address.

After you're done be sure to confirm the changes by sending a test e-mail using the `Send test email...` button.

The Project Presentation

After successfully deploying the online store, Mr. McDew asks you to meet at his office to discuss the new features. At 5am. Knowing he's a stickler for timeliness, you show up at 4:45am.

Todd McDew (TM): Well, well, well, look who finally showed up. Good morning, Mr. Physics.

Mr. Physics (that's you): Physics sir? I'm a programmer, not a physicist.

TM: Keep the sassy remarks to yourself son, it's all science. I see you've successfully deployed the online store. Congratulations. Are we able to sell a downloadable version of the lawncare guide I've been working on? Did anybody tell you I'm Shakespeare with a green thumb, son? My writing is pure poetry.

MP: So I've heard, sir. Yes the store supports downloadable products of all types; PDFs, MP3s, and even videos can be purchased and downloaded.

TM: We don't have to e-mail or put the products on a DVD?

MP: No sir, the entire post-purchase process is completely automated. Customers can download the products directly to their computer.

TM: What about our line of cutting-edge rakes?

MP: You can sell those too. The customer's shipping address is available both within the Stripe interface and via a restricted order management interface in the administration console.

TM: Order management interface? What if we have to issue a refund?

MP: The Stripe console already offers a point-and-click solution for issuing refunds so in the interests of time we didn't recreate that feature in the local order manager, however the API supports refunds so these capabilities could be easily added at some point in the future.

TM: I like the way you think, son. We'll expand only when necessary. OK, let's move on to the final subscriptions phase. I'd like customers to be able to purchase a subscription to one of our famous lawn care service plans. Patty has all of the details.

MP: OK great sir, I'll contact you once the subscription feature is ready for review.

Summary

Congratulations, Mr. McDew continues to be impressed with your ability to deliver as promised. The final desired subscriptions feature is going to be quite a challenge though, so put on some sun screen and grab your favorite rake; it's time to get our hands really dirty!

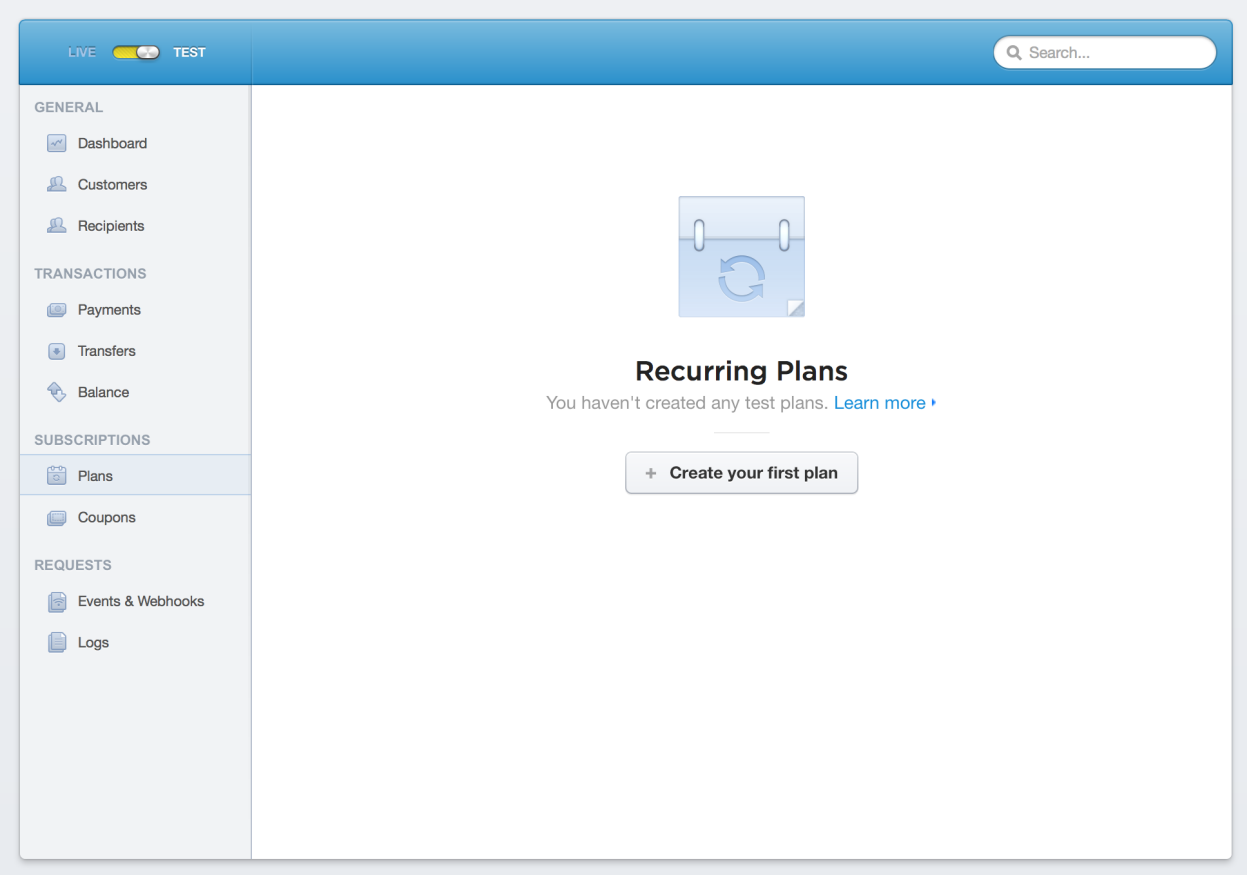
Chapter 5. Selling Subscriptions

Product orders have been rolling in, and Mr. McDew's ambitions continue to grow along with the company's newfound revenue stream. He would like to begin offering an array of subscriptions in which the customer would pay a monthly recurring fee in exchange for a variety of multi-seasonal services such as weed prevention, fertilization, aeration, leaf removal, and snow plowing. By offering year-round services Mr. McDew hopes to stabilize the company's inconsistent, seasonally-based revenue stream and additionally attract new customers interested in a more holistic approach to lawn and property care.

In this chapter you'll learn all about Stripe's recurring billing service, adding a few of these subscription-based services to the company website. We'll begin by defining the desired subscription plans within the Stripe administration console.

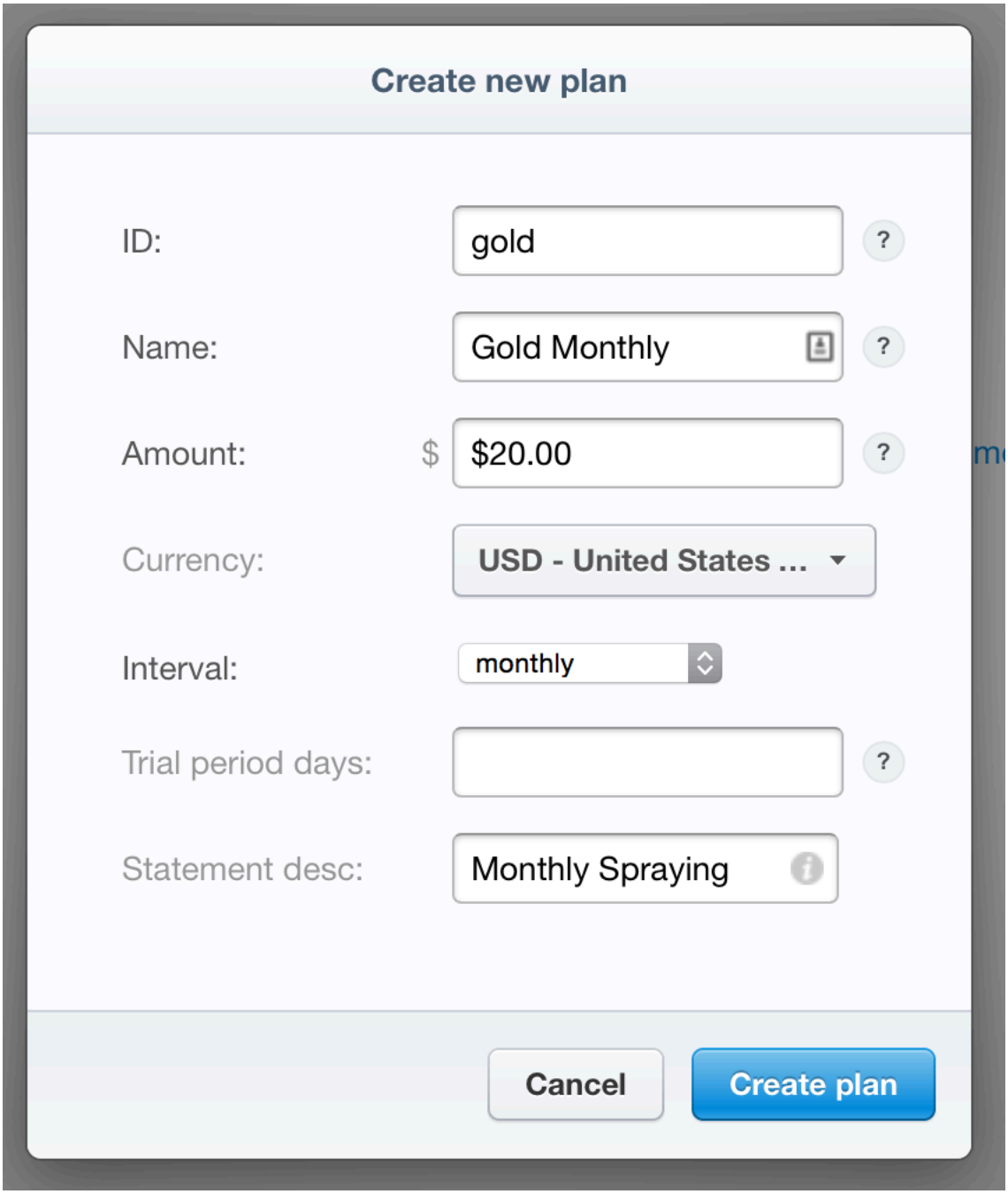
Defining the Subscription Plans

Selling subscriptions through Stripe involves a bit of additional work not otherwise required for product-based offerings. Notably, prior to integrating subscription plans into your website you'll need to define the plans within the Stripe administration console. To do so, sign in to your Stripe dashboard and in the left navigation you'll find a section named "Subscriptions". Click the `Plans` link to create a few plans (see below screenshot).



Stripe Subscriptions

Click the button labeled `Create Your First Plan` and fill out the form presented in the below screenshot:



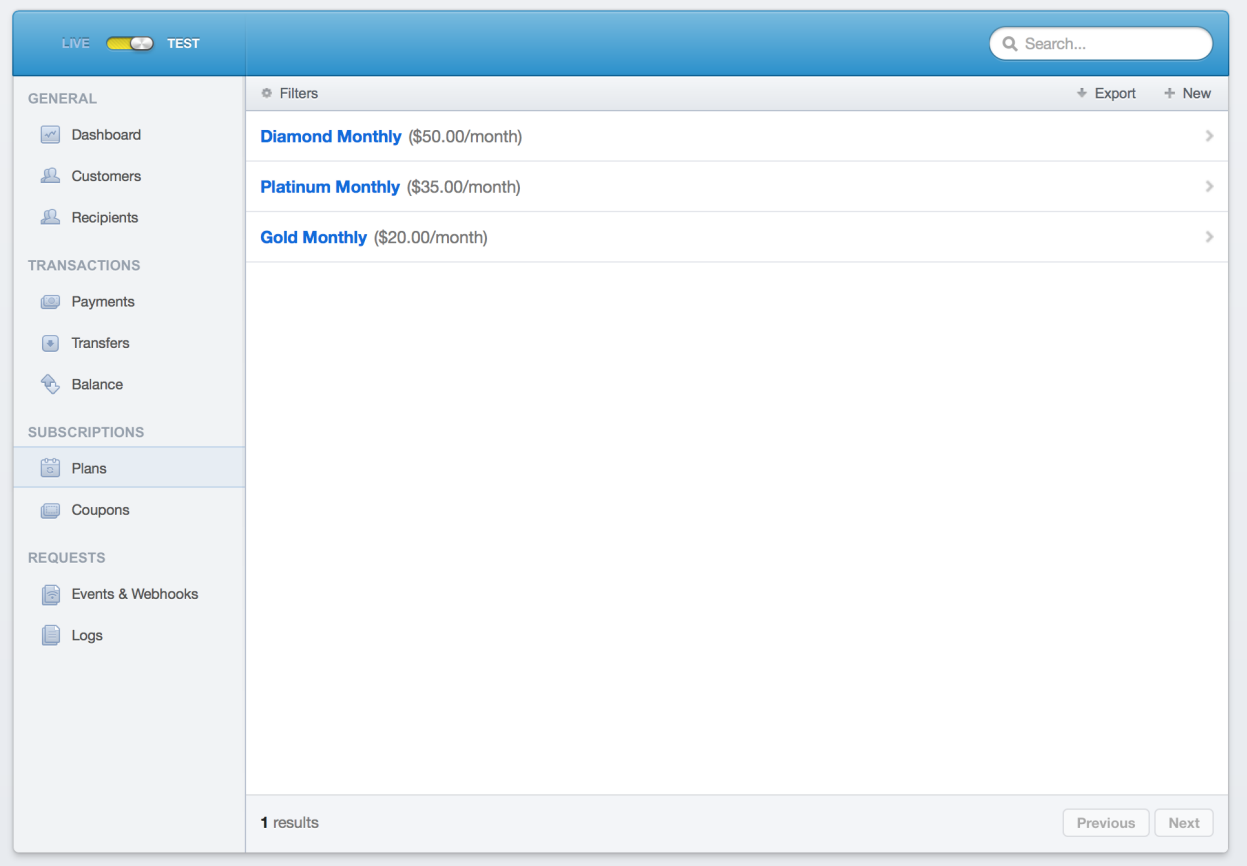
Creating a Stripe Plan

In this example we've created a plan named *Gold Monthly* which is available at a cost of $20 per month, with no trial days. Prior to creating your plans you'll want to spend some time finalizing the plan id (SKU) and price, because you can't change these values in any way after the plan has been created!

Because we are selling lawn care maintenance we will not need to offer trials. If your application

offers trials then you can set the number of days a trial lasts here. While we won't be discussing trials (at least in the book's current iteration), integrating them using Cashier is easy enough; see the Cashier documentation[58] and Stripe Integration Guide[59] for more details.

If you're following along with the steps discussed here, go ahead and create a few more plans for testing purposes. As you can see in the below screenshot we've created three subscriptions in total, including a Gold, Platinum, and Diamond plan.



List of Stripe Subscriptions

Integrating Plans into the Website

In order to provide users with a way to select a plan, let's create a controller and associated view so they can compare the different plans and pick the one that best suits their needs. Begin by creating a new controller named `SubscriptionsController`:

[58] http://laravel.com/docs/master/billing
[59] https://stripe.com/docs/guides/subscriptions

```
php artisan make:controller SubscriptionsController --plain
Controller created successfully.
```

We passed the `plain` flag along because currently this controller only needs to render a view with the list of plans. Next we'll add a route so we can see this page in the browser. Open `app/Http/routes.php` and add the following route:

```
Route::get('plans', 'SubscriptionsController@index');
```

Next, we need to ensure only logged in users can visit this controller. In previous chapters we used the `admin` middleware directly in our routes file. We'll use a different approach here though, adding the middleware directly to the controller class by creating a new `__construct` method:

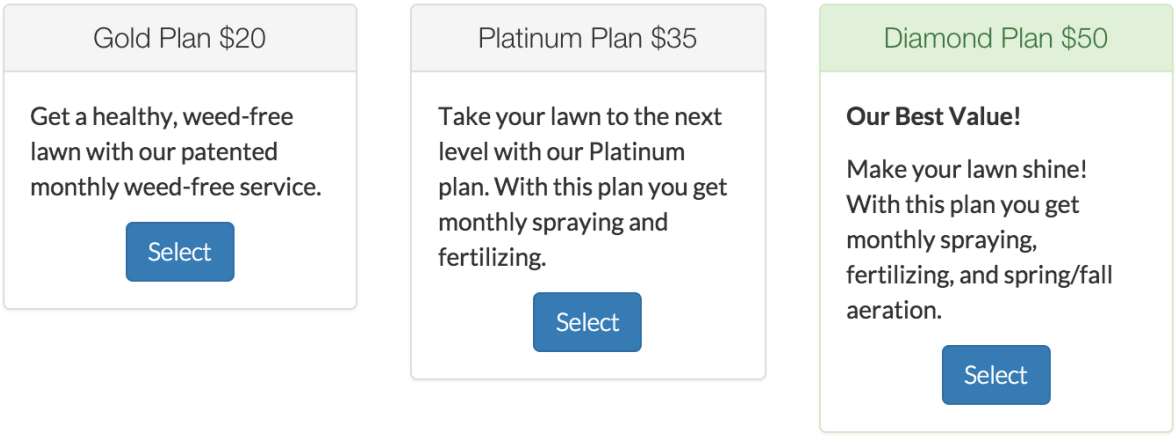
```
public function __construct()
{
    $this->middleware('auth');
}
```

The reason for registering middleware using the controller instead of the route is that this particular controller will do double-duty in the sense it will handle both user subscriptions and invoice management. These two features will use separate routes and therefore to eliminate the middleware registration redundancy in the routes file we can instead register the middleware directly within the controller.

Next, we'll create an `index` method and return a view. Open `Http/Controllers/SubscriptionsController.php` and add the following method:

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

Now let's create the `subscriptions.index` view file referenced by the method. This will display a set of panels containing information about the three available plans (see below screenshot).



Pricing Panels

To add the plans, create a new directory named `subscriptions` inside `resources/views/subscriptions/`, and inside it create a file named `index.blade.php`, pasting in the following code:

```
@extends('app')
@section('intro')
  <div class="intro">
    <div class="container">
      <h1>Make your neighbors envious!</h1>
      <p>Let the professionals at We Dew Lawns, Inc. service your lawn.</p>
    </div>
  </div>
@endsection
@section('content')
    <div class="col-xs-12 col-sm-4">
        <div class="panel panel-default">
            <div class="panel-heading">
                <h3 class="panel-title text-center">Gold Plan $20</h3>
            </div>
            <div class="panel-body">
                <p>Get a healthy, weed-free lawn with our patented
                monthly weed-free service.</p>
                <div class="text-center">
                    <a class="btn btn-primary"
                        href="/plans/subscribe/gold">
                        Select
                    </a>
                </div>
            </div>
```

```
26          </div>
27      </div>
28      <div class="col-xs-12 col-sm-4">
29          <div class="panel panel-default">
30              <div class="panel-heading">
31                  <h3 class="panel-title text-center">Platinum Plan $35</h3>
32              </div>
33              <div class="panel-body">
34                  <p>Take your lawn to the next level with our Platinum plan.
35                  With this plan you get monthly spraying and fertilizing. </p>
36                  <div class="text-center">
37                      <a class="btn btn-primary"
38                          href="/plans/subscribe/platinum">
39                          Select
40                      </a>
41                  </div>
42              </div>
43          </div>
44      </div>
45      <div class="col-xs-12 col-sm-4">
46          <div class="panel panel-success">
47              <div class="panel-heading">
48                  <h3 class="panel-title text-center">Diamond Plan $50</h3>
49              </div>
50              <div class="panel-body">
51                  <p><b>Our Best Value!</b></p>
52                  <p>Make your lawn shine! With this plan you get monthly
53                  spraying, fertilizing, and spring/fall aeration. </p>
54                  <div class="text-center">
55                      <a class="btn btn-primary"
56                          href="/plans/subscribe/diamond">
57                          Select
58                      </a>
59                  </div>
60              </div>
61          </div>
62      </div>
63  @endsection
```

Once the user selects the desired plan, we will redirect them to a form to enter their credit card details. Just like before this will require a route, a controller method, and a view. Let's first add the route:

```
Route::get('plans/subscribe/{planId}', 'SubscriptionsController@subscribe');
```

Now we need to add two new methods to the `SubscriptionsController.php` class:

```
public function subscribe($planId)
{
    if ($this->planNotAvailable($planId))
    {
        return redirect()->route('plans');
    }

    return view('subscriptions.form', compact('planId'));
}

protected function planNotAvailable($id)
{
    $available = ['gold', 'diamond', 'platinum'];
    if ( ! in_array($id, $available))
    {
        return true;
    }
    return false;
}
```

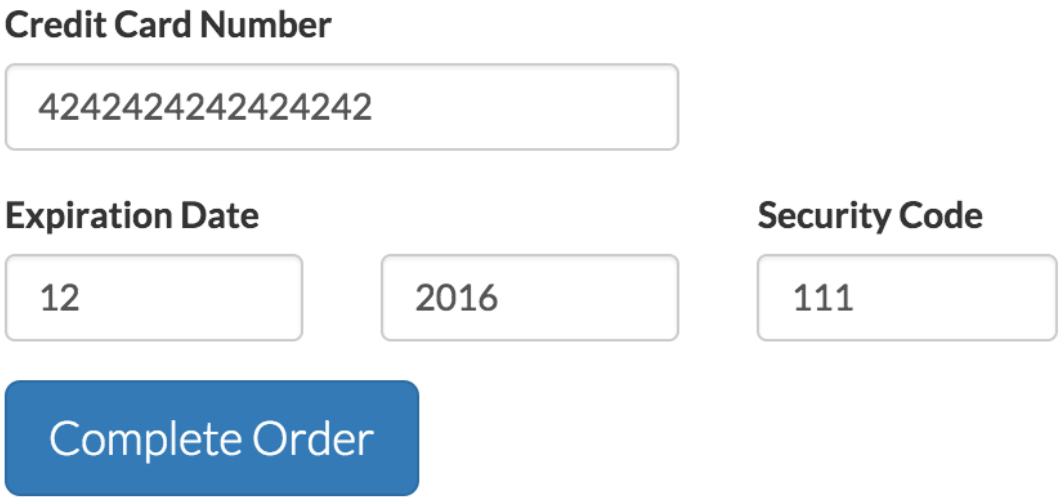
The `subscribe` method accepts a plan ID which is passed along when the user clicks one of the `Subscribe` buttons found in the pricing panels. It uses a protected method named `planNotAvailable` also found in the above snippet, which provides a simple way of verifying the plan id is one we support. If the plan is not found, it then redirects back to our previous step for choosing a new plan. Obviously in more complicated implementations involving a larger number of plans you might manage this in a different fashion, perhaps storing the subscription information in a database table or configuration file, but for our purposes this simple array-based approach will work fine.

Let's create a new view file at `resources/views/subscriptions/subscribe.blade.php`. This view is going to contain our credit card form.

```
@section('content')
  <div class="payment-errors alert alert-danger"
    style="display: none;">
  </div>
  {!! Form::open([
    'route' => 'plans.process',
    'class' => 'form',
    'id' => 'purchase-form'
  ]) !!}
    <input type="hidden" name="plan_id" value="{{ $planId }}" id="plan_id">
    <div class="form-group">
      <div class="row">
        <div class="col-xs-12">
          <label for="card-number" class="control-label">
            Credit Card Number
          </label>
        </div>
        <div class="col-sm-4">
          <input type="text"
            class="form-control"
            id="card-number"
            placeholder="Valid Card Number"
            required autofocus data-stripe="number"
            value="
            {{ App::environment() == 'local' ? '4242424242424242' : '' }}
            ">
        </div>
      </div>
    </div>
    <div class="form-group">
      <div class="row">
        <div class="col-xs-4">
          <label for="card-month">Expiration Date</label>
        </div>
        <div class="col-xs-8">
          <label for="card-cvc">Security Code</label>
        </div>
      </div>
      <div class="row">
        <div class="col-xs-2">
          <input type="text" size="3"
            class="form-control"
```

```
43              name="exp_month"
44              data-stripe="exp-month"
45              placeholder="MM"
46              id="card-month"
47              value="{{ App::environment() == 'local' ? '12' : '' }}"
48              required>
49          </div>
50          <div class="col-xs-2">
51            <input type="text" size="4"
52              class="form-control"
53              name="exp_year" data-stripe="exp-year"
54              placeholder="YYYY" id="card-year"
55              value="{{ App::environment() == 'local' ? '2016' : '' }}"
56              required>
57          </div>
58          <div class="col-xs-2">
59            <input type="text"
60              class="form-control" id="card-cvc"
61              placeholder=""
62              size="6"
63              value="{{ App::environment() == 'local' ? '111' : '' }}"
64              >
65          </div>
66        </div>
67      </div>
68      <div class="center-block form-actions">
69        <button type="submit" class="submit-button btn btn-primary btn-lg">
70          Complete Order
71        </button>
72      </div>
73    {!! Form::close() !!}
74  @endsection
```

Here is a screenshot of what this view looks like in the browser:



Credit Card Form

This is a lot of code and let's step through all the important parts that differ than the other views we've created throughout the book. The first relevant part is the payment errors div:

```
<div class="payment-errors alert alert-danger"
  style="display: none;"></div>
```

We will use this to display any errors returned from the Stripe JavaScript validation. Because a `div` is a block element, and we are using Bootstrap's `alert-danger` class it will present be a glaring red box. However by setting the style to `display: none` it will be hidden from view unless an error occurs.

Another difference in this form compared to all our previous is we are not using the Laravel form helpers. **As was discussed in previous chapters, for security reasons we do not want any credit card information included in the posted data**, and by creating the form manually we can be certain the sensitive form fields do not contain a `name` attribute. This ensures the browser will ignore these fields. This means you no longer need to worry about redacting logs, encrypting cardholder details, or other requirements pertaining to PCI compliance.

Finally we added a default value to each form field:

```
value="{{ App::environment() == 'local' ? '4242424242424242' : '' }}"
```

While you are developing an application, you will want to test the checkout routine multiple times. With this simple check in place, the valid test card number will always be inserted and ready for submission, saving you some time in the process.

Next we'll integrate the JavaScript necessary to submit this form directly to Stripe's servers. As was the case with the last chapter, Stripe will return a credit card token which we can then use to safely charge the card.

Integrating Stripe Into the Subscription Flow

With the page in place, we'll next add the logic required to process the form data and subscribe the user to their chosen plan. First let's add the JavaScript used to submit the form data to Stripe. There are several ways in which this can be accomplished, with perhaps the easiest involving making the JavaScript available specifically to the associated view. Because this JavaScript will not be used in any other part of the website we'll do exactly this in order to ensure the code isn't unnecessarily loaded into other parts of the website.

Typically JavaScript should go in the footer of your HTML just above the closing `</body>` tag. To implement this, we can utilize a new Blade section. Open the `resources/views/app.blade.php` and find the closing `</body>` tag. Next add in the `yield` statement just above it like this:

```
    @yield('footer_js')
</body>
</html>
```

Now open the `resources/views/subscriptions/form.blade.php` again and add the following JavaScript:

```
@section('footer_js')
<script type="text/javascript" src="https://js.stripe.com/v2/"></script>
<script>
  Stripe.setPublishableKey('{{ env('STRIPE_API_PUBLIC') }}');
  jQuery(function($) {
    $("#card-number").focusout(function() {
      var el = $(this);
      if ( ! Stripe.validateCardNumber(el.val())) {
        el.closest(".form-group").addClass("has-error");
      } else {
        el.closest(".form-group").removeClass("has-error");
      }
    });
    $("#card-cvc").focusout(function() {
      var el = $(this);
      if ( ! Stripe.validateCVC(el.val())) {
        el.closest("div").addClass("has-error");
      } else {
        el.closest("div").removeClass("has-error");
      }
    });
    $('#purchase-form').submit(function(e) {
```

```
23        $('.submit-button').prop('disabled', true);
24        var $form = $(this);
25        $form.find('.payment-errors').hide()
26        Stripe.card.createToken({
27          number: $form.find('#card-number').val(),
28          cvc: $form.find('#card-cvc').val(),
29          exp_month: $form.find('#card-month').val(),
30          exp_year: $form.find('#card-year').val()
31        }, stripeResponseHandler);
32        return false;
33      });
34    });
35    var stripeResponseHandler = function(status, response) {
36      var $form = $('#purchase-form');
37      var $errors = $('.payment-errors');
38      // Reset any errors
39      $errors.text("");
40      if (response.error) {
41        $errors.text(response.error.message).show();
42        $form.find('button').prop('disabled', false);
43      } else {
44        var token = response.id;
45        $form
46          .append($('<input type="hidden" name="stripe_token" />')
47          .val(token));
48        $form.get(0).submit();
49        $form.find('button').html('Processing...');
50      }
51    };
52  </script>
53  @endsection
```

That is a lot of JavaScript so let's step through it piece by piece. At the top, we are referencing the externally hosted `Stripe.js` library and then setting our public API key:

```
1  Stripe.setPublishableKey('{{ env('STRIPE_API_PUBLIC') }}');
```

Next you'll find two validation methods involving jQuery's `focusout` function. These both utilize Stripe functions to validate both the credit card number and the CVC code. If it fails, the forms fields will be highlighted red, informing the user that something is wrong. This happens on the client-side, meaning the user will be notified of these problems before submitting the form.

Next we process the form using the following code:

```
$('#purchase-form').submit(function(e) {
    $('.submit-button').prop('disabled', true);
    var $form = $(this);
    $form.find('.payment-errors').hide()
    Stripe.card.createToken({
        number: $form.find('#card-number').val(),
        cvc: $form.find('#card-cvc').val(),
        exp_month: $form.find('#card-month').val(),
        exp_year: $form.find('#card-year').val()
    }, stripeResponseHandler);
    return false;
});
```

Once the submit button is clicked, we disable the button to prevent duplicate clicks, hide any errors, and finally call Stripe's `createToken` method. This accepts an object containing the credit card details and additionally a second parameter referring to a `stripeResponseHandler` callback.

The `stripeResponseHandler` function handles both failed attempts and successful ones from the Stripe API.

If credit card validation fails, we will display the error returned by Stripe within the hidden `div`, forcing it to show, and setting the submit to enabled.

```
$errors.text(response.error.message).show();
$form.find('button').prop('disabled', false);
```

If the credit card is accepted we append a new hidden input named `stripe_token` with Stripe's token, then submitting the form using the following statement:

```
$form.get(0).submit();
```

The final step in subscribing the user is to retrieve the card token and send the final request off to Stripe. Create a new `process` method in the `SubscriptionsController`:

```
public function process(Request $request)
{
    $planId = $request->get('plan_id');
    if ($this->planNotAvailable($planId)) {
        return redirect()->back()->withErrors('Plan is required');
    }
    $user = Auth::user();
    $user->subscription($planId)->create($request->get('stripe_token'), [
        'email' => $user->email,
        'metadata' => [
            'name' => $user->name,
        ],
    ]);
    return redirect('invoices');
}
```

Like the earlier `index` action, here we'll again check to ensure the submitted plan exists, and if not redirect back to the form and display an error. If the plan does exist, we'll create the subscription using the Cashier-enhanced `User` model:

```
$user = Auth::user();
$user->subscription($planId)->create($request->get('stripe_token'), [
    'email' => $user->email,
    'metadata' => [
        'name' => $user->name,
    ],
]);
```

The user subscription method accepts the `$planId` and uses the `create` method to record the subscription within the Stripe service. The `create` method accepts the Stripe token, and additionally an array of customer details. In this example, we pass in the user's email, and an optional set of key/value pairs containing customer metadata. The metadata is free form, meaning you can pass any data along which can later be referenced inside the Stripe administration panel.

At this point a user will be able to subscribe to a plan. However, there are plenty of other administrative features we could add to the site. We'll spend the remainder of this chapter exploring a host of administrative features, including providing downloadable invoices, a facility for upgrading/downgrading subscriptions, the ability to cancel a subscription, and the ability to use a coupon to receive a subscription discount.

Integrating Subscription Management Features

Mr. McDew has been steadfast on providing subscribers with an easy online solution for accessing invoices, changing plans, and even canceling their subscription. He is worried if customers can't

autonomously manage these matters then support requests will skyrocket. In this section we'll discuss all of these features, beginning with invoice accessibility.

Providing Downloadable Invoices

Customers appreciate the ability to download current and past invoices. In this section we'll show you how to do so. Begin by opening `app/Http/routes.php` and adding the following route definition. We'll use this route as a conventional way for customers to access their invoices:

```
Route::get('invoices', [
    'as' => 'invoices', 'uses' => 'SubscriptionsController@invoices'
]);
```

Now we need to add a new `invoices()` method to the `Http/Controllers/SubscriptionsController.php` file:

```
public function invoices()
{
    $user = Auth::user();
    return view('subscriptions.invoices', compact('user'));
}
```

In this method, we are assigning the `Auth::user()` to a `$user` variable and assigning it to the view. With the user available we can then load in a list of all their invoices using a convenience method made available via Cashier. The following code snippet demonstrates how such a summary might be presented to the user:

```
<table class="table">
    @foreach ($user->invoices() as $invoice)
        <tr>
            <td>
                Paid {{ $invoice->dollars() }} on
                <small>{{ $invoice->dateString() }}</small>
            </td>
            <td>
                <a href="/invoices/download/{{ $invoice->id }}">
                    Download Receipt
                </a>
            </td>
        </tr>
    @endforeach
</table>
```

The following screenshot presents an example of what the resulting page looks like:

Your Invoices

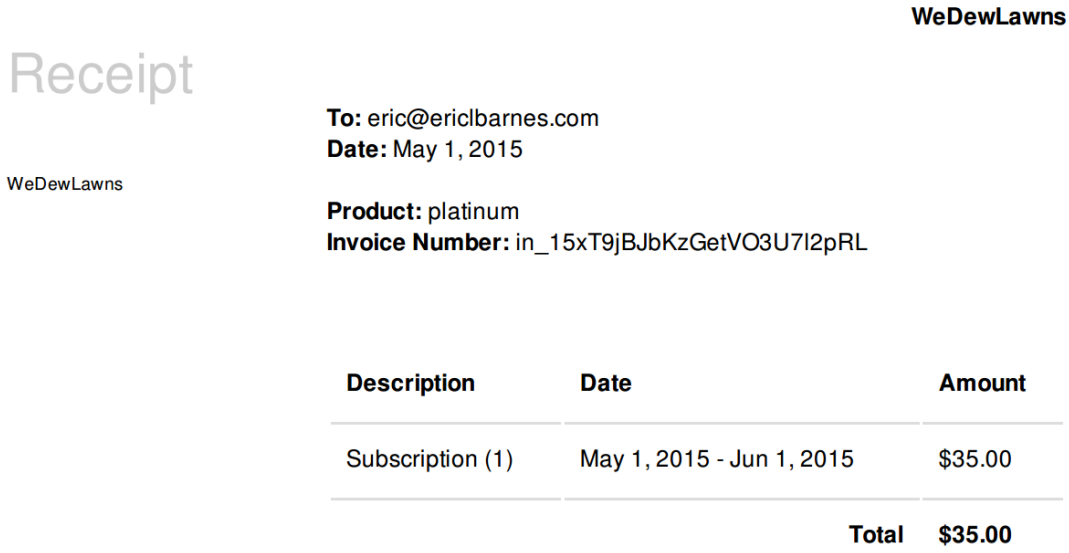
When the user clicks the download receipt link it should initiate a file download. Open the `SubscriptionsController.php` file again and add the new route:

```
Route::get('invoices/download/{id}', [
    'uses' => 'SubscriptionsController@downloadInvoice'
]);
```

Next add a new `downloadInvoice()` method to the `Http/Controllers/SubscriptionsController.php` file:

```
public function downloadInvoice($id)
{
    return Auth::user()->downloadInvoice($id, [
        'vendor'  => 'WeDewLawns',
        'product' => Auth::user()->stripe_plan,
    ]);
}
```

This utilizes Laravel Cashier's `downloadInvoice` method and creates a PDF that looks like that found in the below screenshot.



WeDewLawns

Receipt

WeDewLawns

To: eric@ericlbarnes.com
Date: May 1, 2015

Product: platinum
Invoice Number: in_15xT9jBJbKzGetVO3U7l2pRL

Description	Date	Amount
Subscription (1)	May 1, 2015 - Jun 1, 2015	$35.00
	Total	**$35.00**

An example invoice

That includes all the important information about the transaction, but it's not very pretty and doesn't carry the company branding. You can include company information by passing additional information into `downloadInvoice`:

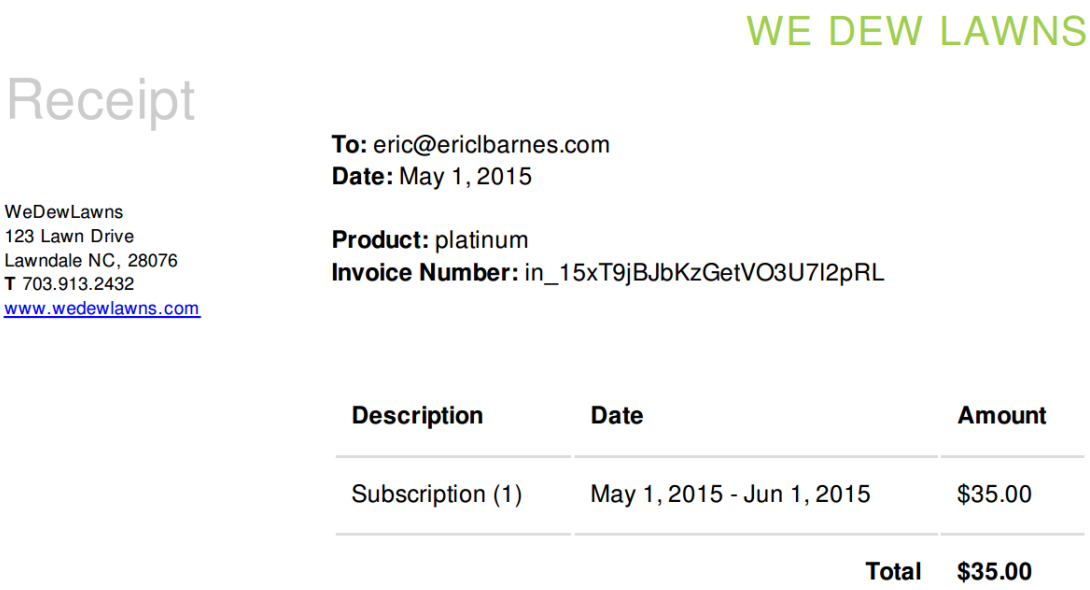
```
1  return Auth::user()->downloadInvoice($id, [
2      'header'  => 'We Dew Lawns',
3      'vendor'  => 'WeDewLawns',
4      'product' => Auth::user()->stripe_plan,
5      'street' => '123 Lawn Drive',
6      'location' => 'Lawndale NC, 28076',
7      'phone' => '703.555.1212',
8      'url' => 'www.wedewlawns.com',
9  ]);
```

Now when the PDF is generating it will include all of company's contact details but it's still rather bland. Let's make the header green to match the website.

To customize this, we need to modify the default view file that ships with Cashier. To access the file we need to utilize an artisan command to move it into our views folder:

```
$ php artisan vendor:publish
Copied Directory [/vendor/laravel/cashier/src/views] To [/resources/views/vendor\
/cashier]
Publishing Complete!
```

After running this command, you can access the file from `resources/views/vendor/cashier/receipt.blade.php` and customize it to suit your specific design needs. The following screenshot presents an example of what's possible.



WE DEW LAWNS

Receipt

To: eric@ericlbarnes.com
Date: May 1, 2015

WeDewLawns
123 Lawn Drive
Lawndale NC, 28076
T 703.913.2432
www.wedewlawns.com

Product: platinum
Invoice Number: in_15xT9jBJbKzGetVO3U7l2pRL

Description	Date	Amount
Subscription (1)	May 1, 2015 - Jun 1, 2015	$35.00
	Total	**$35.00**

A Customized Invoice

Swapping Subscriptions

There will be times when a customer desires to upgrade or downgrade their subscription plans. Cashier provides a simple way of implementing this by including the `User` model's `swap()` method. In this section we'll show you an easy way to implement this feature. Open the `app/Http/routes.php` file and add a new route:

```
Route::post('plans/swap', [
  'as' => 'plans.swap', 'uses' => 'SubscriptionsController@swapPlans'
]);
```

Now open `app/Http/Controllers/SubscriptionsController.php` and add a new `swapPlans` method:

```
public function swapPlans(Request $request)
{
  $planId = $request->get('plan_id');
  if ($this->planNotAvailable($planId)) {
      return redirect()->back()->withErrors('Plan is required');
  }
  Auth::user()->subscription($planId)->swap();
  return redirect()->back()->withMessage('Plan changed!');
}
```

Back in our `resources/views/subscriptions/invoices.blade.php` view we just need to add a new form and a select list for the user to choose the new plan.

```
{!! Form::open(['route' => 'plans.swap', 'class' => 'form-horizontal']) !!}
  <select name="plan_id" class="form-control" id="plan_id">
    <option value="gold">Gold / $20 per month</option>
    <option value="platinum">Platinum / $35 per month</option>
    <option value="diamond">Diamond / $50 per month</option>
  </select>
  <button type="submit" class="btn btn-default">Swap Plans</button>
{!! Form::close() !!}
```

Canceling a Subscription

Despite stellar lawn care service and competitive pricing, customers will occasionally desire to cancel their subscription. Implementing a solution for customer-driven subscription cancellation is practically identical to that used for subscription upgrades/downgrades, involving a new route, a controller method, and a form. Begin by adding the following route to the `routes.php` file:

```
Route::post('plans/cancel', [
  'as' => 'plans.cancel', 'uses' => 'SubscriptionsController@cancelPlan'
]);
```

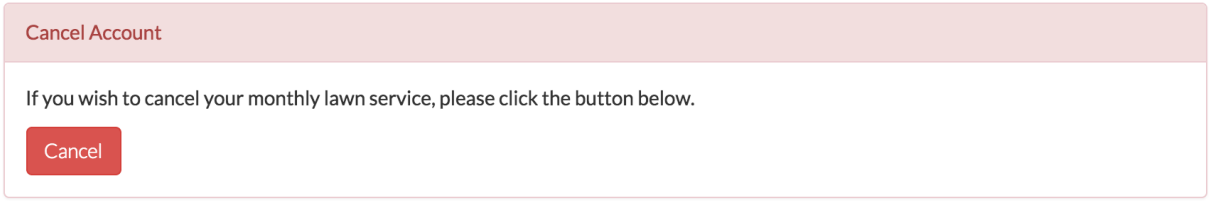
Next, return to the `SubscriptionsController.php` add the following `cancelPlan` method:

```
public function cancelPlan()
{
  Auth::user()->subscription()->cancel();
  return redirect('invoices')->withMessage('Your plan has been cancelled');
}
```

Finally, we just need to add a new form to the `resources/views/subscriptions/invoices.blade.php` view:

```
{!! Form::open(['route' => 'plans.cancel', 'class' => 'form-inline']) !!}
  <p>If you wish to cancel your monthly lawn service, please
  click the button below.</p>
  <button type="submit" class="btn btn-danger">Cancel</button>
{!! Form::close() !!}
```

The following screenshot presents an example of what this would look like after applying a Bootstrap panel style:

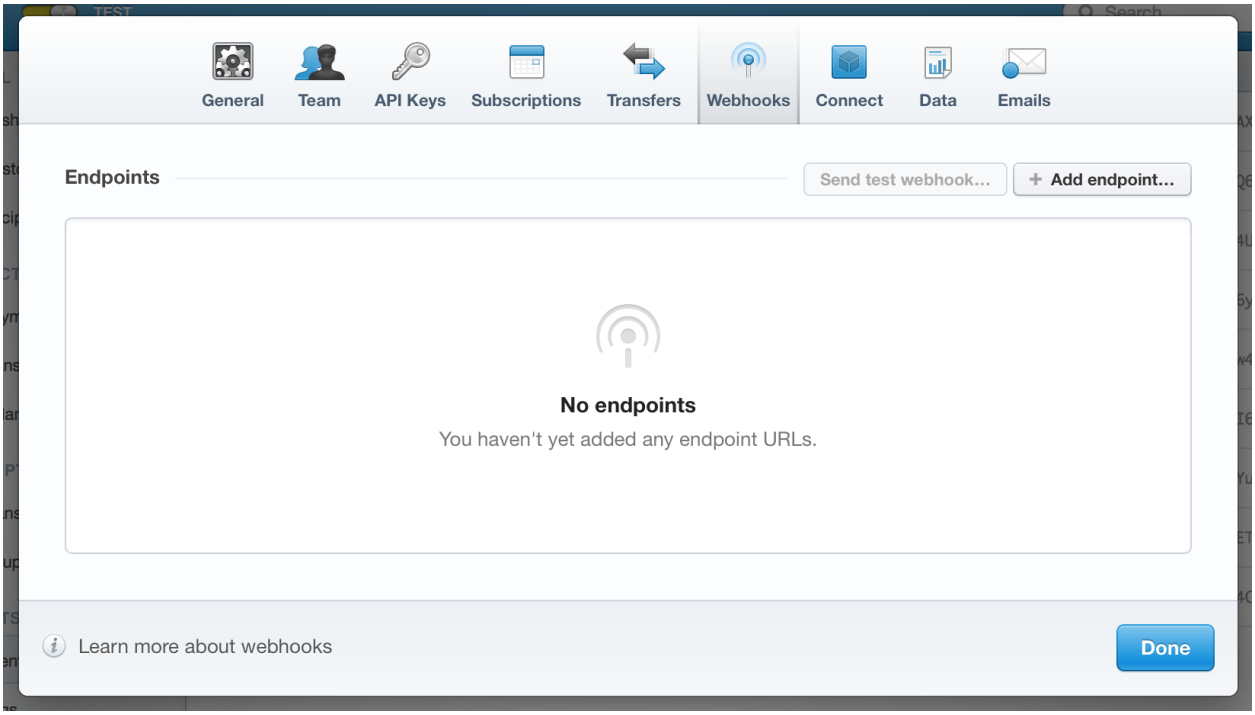


Cancel Account Form

Now that a subscriber can fully manage their account it's time to configure webhooks for those situations when credit cards expire, payment fails, or other actions are initiated from within the Stripe service.

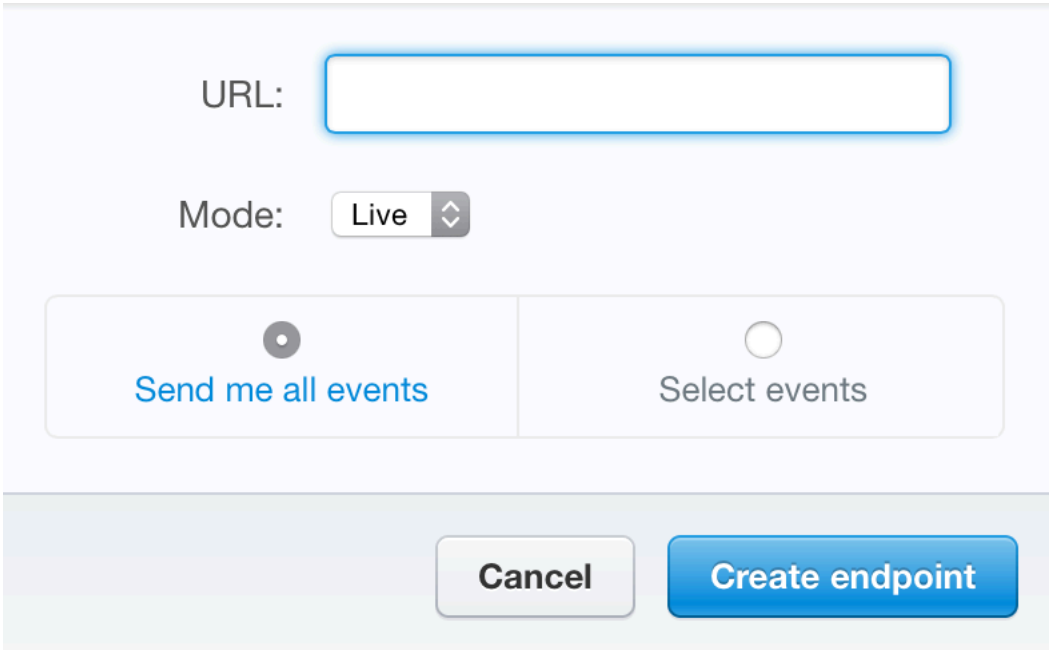
Listening to Events with Webhooks

Webhooks are a way for your site to be notified when events happen within Stripe, such as when a new customer signs up for a subscription, or an existing customer credit card expires. In both of these cases Stripe can be configured to contact your webhook in order to inform your server of these events. To configure a webhook return to the Stripe administration console and click the `Your account` link located at the top right of the page, and then select `Account Settings` in the menu. From there click the `Webhooks` tab (see the below screenshot):



Stripe Webhooks

Click the `Add endpoint...` button and the following new window will appear:



Creating a Webhook endpoint

On this screen we have the option of entering a URL, the mode, and a choice between all events or selected events. For the URL enter `example.com/stripe/webhook`, replacing `example.com` with your

actual domain. For mode select `Test` and finally select `Send me all events` to ensure that all events are sent to your webhook, which affords you the convenience of not having to return to the Stripe administration console every time you'd like to expand the listening capabilities of your webhook.

Processing a Webhook Event

Recall when we created the Stripe webhook we entered the url `/stripe/webhook` and we will need to configure a route. Open `app/Http/routes.php` and add this new route:

```
1  Route::post('stripe/webhook', 'StripeController@handleWebhook');
```

Now let's create the new `StripeController` controller, which we'll use to manage the webhook processor. Open up the terminal and run:

```
1  php artisan make:controller StripeController --plain
2  Controller created successfully.
```

We passed the `--plain` flag because we will only need a single method to consume the posted data from Stripe. Next we need to adjust this newly created controller. Open `StripeController.php` and you will see this empty class:

```
1  <?php namespace App\Http\Controllers;
2
3  use App\Http\Requests;
4  use App\Http\Controllers\Controller;
5
6  use Illuminate\Http\Request;
7
8  class StripeController extends Controller {
9
10     //
11
12 }
```

This needs to be adjusted to extend `\Laravel\Cashier\WebhookController`:

```
1  class StripeController extends \Laravel\Cashier\WebhookController {
```

By extending the `WebhookController` we'll have full access to Cashier's event system so we can create our own webhook listener methods.

In our last meeting Mr. McDew mentioned how he would like to receive an email every time a charge fails. To implement this feature Stripe offers a webhook named `invoice.payment_failed` that will notify us on any declined payments. Because Cashier only listens for the `customer.subscription.deleted` event we will need to create a new method to listen for this.

Cashier follows a specific pattern for events. For example if we are wanting to listen for `invoice.payment_failed` then we would create a method named `handleInvoicePaymentFailed`. Prefixing with handle, removing the periods, and upper casing the letters.

Taking this naming convention create a new method in the `StripeController` controller:

```
1  public function handleInvoicePaymentFailed($payload)
2  {
3      $billable = $this->getBillable(
4        $payload['data']['object']['customer']
5      );
6
7      if ($billable) {
8          \Mail::send('emails.failed_charge',
9            compact('billable'), function($message)
10             {
11                     $message->to(env('MAIL_FROM'), env('MAIL_NAME'));
12                 $message->subject('WeDewLawns.com Payment Failed');
13             });
14     }
15
16     return new Response('Webhook Handled', 200);
17 }
```

What this method does is first call `$this->getBillable()` which retrieves the user record from the database. Then as long as it's found we send out an email using a new `emails.failed_charge` template which should be created at: `resources/views/emails/failed_charge.blade.php`.

Disabling CSRF Protection

For security reasons Laravel has global CSRF protection enabled which means no data can be posted without including a randomly generated token (the generation process is managed automatically by Laravel). In most cases this is a perfectly reasonable security procedure but it introduces a problem with web hooks since they originate from the Stripe server, meaning when the Stripe controller is contacted via `POST` Laravel will complain because the request is not accompanied by the CSRF token. You can however bypass this CSRF check by modifying the `app/Http/Middleware/VerifyCsrfToken.php` file. After opening the file add the following code to the `handle()` method:

```
if ($request->is('stripe/webhook')) {
    return $this->addCookieToResponse($request, $next($request));
}
```

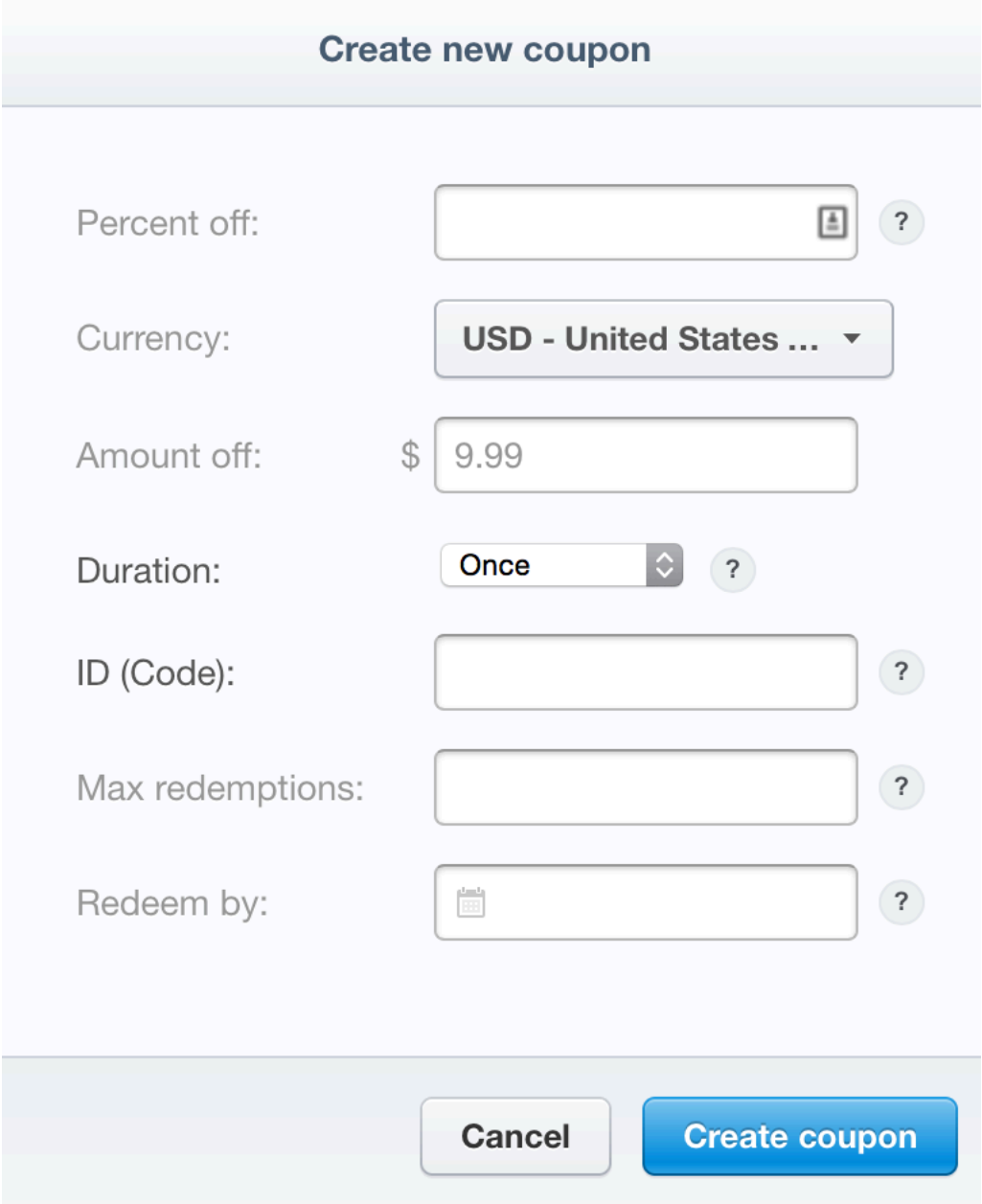
This will cause Laravel to completely bypass the CSRF check if the request is posting data to the `stripe/webhook` routes. Incidentally, at the time of this writing, Laravel 5.1 hasn't yet been released but it will include an option for bypassing this using an `except` property on the `VerifyCsrfToken` class:

```
protected $except = [
    'stripe/webhook/*'
];
```

That will be the recommended way once 5.1 is officially released.

Creating Coupons

To generate interest in the new lawn care subscriptions Mr. McDew would like to include a few coupons in the restricted area created in Chapter 2. Fortunately, coupon integration is very straightforward when using Cashier. To begin you'll need to visit the Stripe administration console and click on the coupons link located in the sidebar. Click "New" and you will see the following form:



Stripe Coupons Form

Let's have a look at the relevant settings:

- **Percent Off** and **Amount Off**: These two fields allow you to define either percent of the total amount off. When creating a new coupon you will only enter a number for the type you are creating.
- **Duration**: This setting allows you to define how the coupon lasts, once applied. The options are once, multi-month, or forever.
- **ID Code**: This is the code the user will enter to take advantage of the discount.

- **Max Redemptions**: If you would like to restrict the coupon to a set number of redemptions.
- **Redeem By**: For short term sales you can set a date here and this coupon will not be available to use after that date.

There are two ways to integrate coupons into the website: at the time of subscribing and after they have subscribed. For the former, you can utilize Cashiers `withCoupon` method:

```
$user->subscription($planId)
  ->withCoupon('code')
  ->create($request->get('stripe_token')
```

If a user already has a subscription you can assign a coupon to their subscription using the `applyCoupon` method as follows:

```
try {
    Auth::user()->applyCoupon('code');
    return redirect()->back()->withMessage('Coupon Applied');
} catch (Exception $e) {
    return redirect()->back()->withError($e->getMessage());
}
```

When applying a coupon it's best to wrap this in a try / catch block because if the user enters an invalid coupon, Stripe will return an exception with the reason for failure.

The Project Presentation

The recurring subscriptions implementation phase is complete and has been successfully deployed to production. Mr. McDew has asked you to meet at the local truck stop diner to discuss the new capabilities. At midnight.

Mr. McDew: Well, well, Susie (Susie is the waitress), look what the cat dragged in. How are we doing, Mr. Techno Wizard? Coffee?

MTW (that's you): No thank you. After all it's midnight, sir. Aren't you going to sleep soon?

TM: Sleep is for the weak, son! I have lawns to mow and dirt to throw. Tell me about the latest changes to the site. Are customers able to subscribe to our new array of lawn care services?

MTW: Yes sir, everything has been implemented and is ready for promotion. We used Stripe's recurring billing services, so all customer payments are conveniently managed using the same service processor.

TM: Can customers upgrade their subscriptions? You know that once they get a taste of this sweet service they won't be able to help themselves, right?

MTW: Yes sir, customers can both upgrade and (gulp) downgrade their service plans. They can also cancel plans using the website.

TM: Cancel plans? Are you trying to bankrupt me, son? I have pitbulls to feed. Anyways, yes I guess this is a good idea, as Patty already has enough to do without having to constantly deal with subscriptions. What about coupons?

MTW: Coupons are ready as well. You can add coupon codes to the restricted authenticated user section of the site, and customers can use them as desired when purchasing a new subscription.

TM: OK, I'll contact you soon as I'd like to start implementing a shopping cart later this month. Right now though, I'm working on a new idea with Susie here, we're going to create a new line of casual clothing for pets. Pets, son! That's where the real money is at.

MTW: Great sir, I'm happy to meet again whenever you are ready.

TM: You're still here? Time for you to skedaddle, I don't want you listening in on our product planning session. Now Susie, where were we…

Summary

In this phase you've created recurring plans in Stripe, integrated subscription processing into the We Dew Lawns website, configured webhooks, and added an array of useful subscription management features. Stay on your programming toes though, because Mr. McDew will soon be contacting you to revise the site in order to implement a shopping cart!

Chapter 6. Integrating a Shopping Cart

Awash in newfound revenue, Mr. McDew has taken to wearing pinstriped suspenders and slicking his hair back Ã la Wall Street's Gordon Gecko. We've created a monster! While he's clearly very happy with WeDewLawns.com, his biggest complaint is the lack of a shopping cart, because customers are required to go through the product purchase process multiple times should they be interested in purchasing more than one product.

This outcome wasn't accidental, because we've preferred to focus on Laravel- and Stripe-related fundamentals so as to ensure everything is working flawlessly before introducing more complicated features. In this chapter we'll change that by adding shopping cart capabilities to the site. By the conclusion of this chapter you'll have successfully integrated a shopping cart into the site, allowing customers to truly shop the site, and conveniently purchase more than one product during a single session.

Creating the Cart Model

Let's get things rolling by creating the model and associated table used to manage the shopping cart contents:

```
$ php artisan make:model Cart -m
Model created successfully.
Created Migration: 2015_06_07_180530_create_cart_table
```

Next, open up the newly created migration and modify the `up()` method to look like this:

```
public function up()
{
  Schema::create('cart', function(Blueprint $table)
  {
    $table->increments('id');
    $table->integer('user_id');
    $table->integer('order_id')->nullable();
    $table->integer('product_id');
    $table->integer('complete')->default(0);
    $table->integer('qty');
```

```
11      $table->decimal('price', 8, 2);
12      $table->timestamps();
13    });
14  }
```

First off, note we've made a rare break from convention in terms of the table name in that the singular form of `cart` is used rather than the plural `carts`. This is simply a preferential matter, implemented with the reason that we refer to a user's "cart" rather than "carts".

Like all tables, the `cart` table includes a unique integer-based automatically incrementing ID column. However the meaning of one or several other columns might not be so obvious, so let's review them now:

- The `user_id` column associates each cart with a user. This initial implementation requires the user to be signed into WeDewLawns.com, and therefore this column will identify each user by their associated primary key found in the `users` table.
- The `order_id` column associates a shopping cart with a completed order once the customer has successfully checked out.
- The `product_id` column identifies the product currently residing in the cart by its `products` table primary key.
- The `complete` column determines whether this shopping cart has been converted into a completed order.
- The `qty` column identifies the quantity of products the user would like to purchase.
- The `price` column identifies the per-unit price of the product found in the cart. Because prices are typically defined using decimal values, the column declaration specifies a `decimal` datatype supporting a maximum of eight digits (known as the *precision*), and a maximum of two digits found to the right of the decimal (known as the *scale*). This means values between `0.00` and `999,999.99` can be stored in this column. Perhaps we're getting a little carried away with the maximum price, however you can easily adjust this downwards to support values between `0.00` and `999.99` by instead defining the column as `decimal('price', 5, 2)`.

After saving the changes, run the migration to create the table:

```
1       $ php artisan migrate
2       Migration table created successfully.
3       Migrated: 2015_06_07_180530_create_cart_table
```

Next, because the `cart` table references several other tables (`users`, `orders`, and `products`), we'll need to configure the associations within the various models. Let's begin with the `Cart` model. We'll present the implemented model next (`app/Cart.php`), followed by a summary of the changes:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Cart extends Model {

protected $table = 'cart';

protected $fillable = ['product_id', 'qty', 'price'];

public function product()
{
    return $this->belongsTo('App\Product');
}

public function user()
{
    return $this->belongsTo('App\User');
}

    }
```

The `$table` property is used to override Laravel's default presumption of a model's corresponding pluralized table name. Because we overrode the migration to instead create a table named `cart`, we need to tell the model to instead look for the `cart` table instead of a `carts` table.

The `$fillable` property allows us to programmatically mass-assign the `product_id`, `qty`, and `price` column values when inserting and updating records found in the `cart` table. This isn't necessary but is instead done to make the code a tad more succinct. Security issues can arise if you assign sensitive column names to `$fillable`, so be sure to consult the Laravel documentation if you're not entirely aware of how this property behaves.

The `product()` method defines the `Cart` model's belongs to association with the `Product` model. Because the `cart` table stores a `products` table foreign key, each record in the `cart` table "belongs to" a record in the `products` table. Similarly, the `user()` method defines the `Cart` model's belongs to association with the `User` model, meaning each record in the `cart` table "belongs to" a record in the `users` table.

Because we'll want to retrieve a cart by way of its owner (a user), you'll also want to define the converse association in the `User` model. Do so by opening the `User` model and adding the following method:

```
public function cart()
{
    return $this->hasMany('App\Cart')
      ->where('complete', 0);
}
```

This will retrieve all of the `cart` table records associated with a given user which have not yet been identified as completed (converted into a successful order).

Creating the Carts Controller

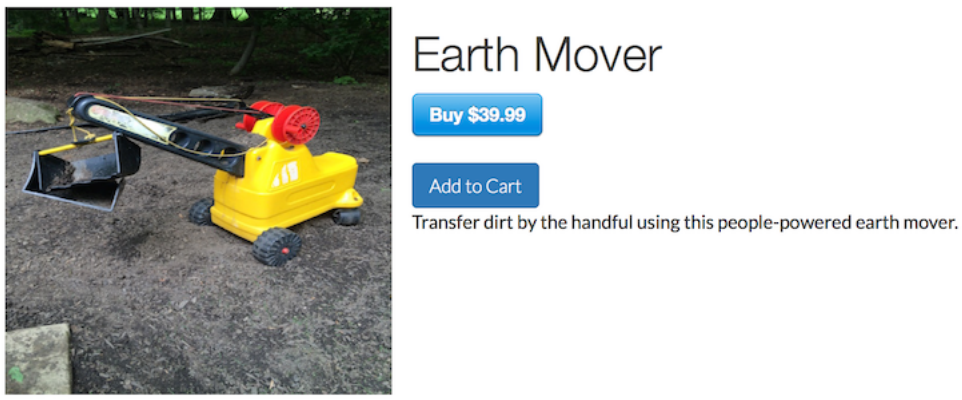
With the `Cart` model and associated table in place, let's next tackle the controller responsible for viewing a cart, and adding and removing products to and from the cart. Begin by generating the `CartsController` controller. Because we won't need all of the routing available to a RESTful controller, let's instead just create a plain controller:

```
$ php artisan make:controller --plain CartsController
Controller created successfully.
```

With the controller created, we'll spend the remainder of this section building out the add, view, remove, and process features, in that order.

Adding Products to the Cart

It seems logical to implement the product addition feature before other capabilities, since without this ability we can't reasonably implement the view, remove, or process features. Fortunately, it is incredibly easy to implement cart product addition. Begin by adding an `Add to Cart` button to the `product/show.blade.php` view, as depicted in the following screenshot:



The Add to Cart button

This button is actually a submit button for a form containing a single hidden field containing the product page's product ID. The form looks like this:

```
        {!! Form::open(['url' => '/cart/store']) !!}
        <input
          type="hidden"
          name="product_id"
          value="{{ $product->id }}"/>
        <button
          type="submit"
          class="btn btn-primary">Add to Cart
        </button>
        {!! Form::close() !!}
```

Note how the `Form::open` method references the URI `/cart/store`. This URI will point to the `CartController` `store` method, so go ahead and add that route now to the `app/Http/routes.php` file:

```
        Route::post('cart/store', 'CartController@store');
```

Next, open the `CartController` class (`app/Http/Controllers/CartController.php`) and add the following `store()` method:

```
        use App\Cart;
        use App\Product;

        ...

  public function store(Request $request)
  {

  $product = Product::find($request->get('product_id'));

  $cart = new Cart([
      'product_id' => $product->id,
      'qty' => $request->get('qty', 1),
      'price' => $product->price,
  ]);

  Auth::user()->cart()->save($cart);
  return redirect('/cart');

  }
```

Although you're certainly free to use a custom form request here, we're using the default (no validation) since the form submits just one field (in addition to the CSRF token) identifying the product ID. We'll use that ID to retrieve the associated product record, and then pass that product ID, the quantity, and the product price into a new `Cart` object. For reasons of simplicity we've set the `qty` default to `1` and will leave it to you as an exercise to add a form field or select box which allows the user to adjust the quantity.

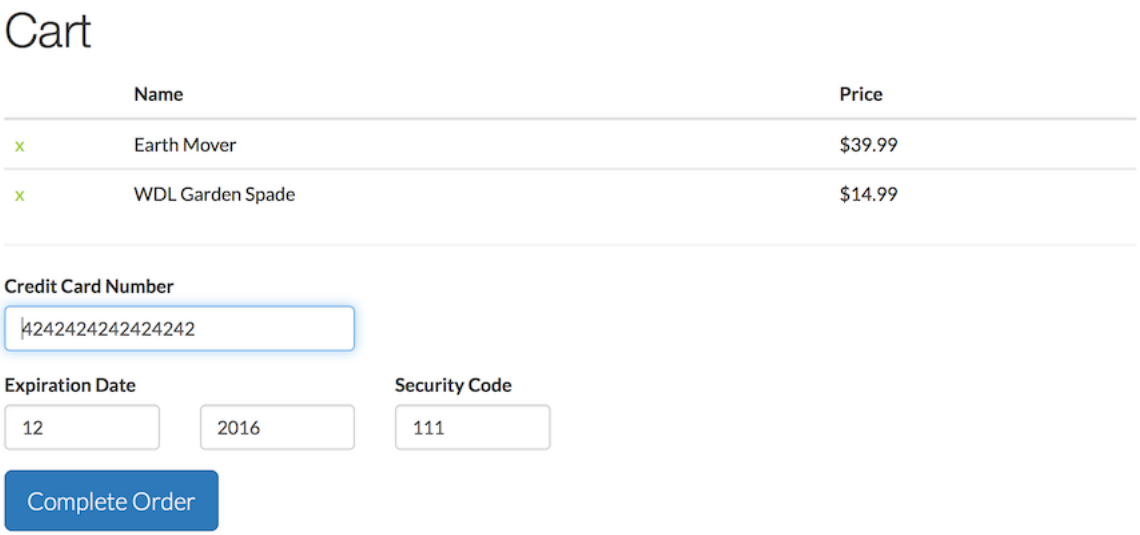
With this `Cart` object created, we use Laravel's convenient model association capabilities to associate it with the currently signed-in user before redirecting the user to the `/cart` URI (which we'll implement in a moment). Because we're using `Auth::user()` without first determining whether the user is already signed in, you'll additionally want to execute the `auth` middleware in the controller constructor:

```
1  public function __construct()
2  {
3      $this->middleware('auth');
4  }
```

If after implementing the code found in this section you navigate to a product view page and click the `Add to Cart` button, the product will indeed be added to your shopping cart however Laravel will subsequently return a `NotFoundHttpException` exception, because the `/cart` URI doesn't exist. Let's implement that next so users can actually see what's in their cart.

Viewing the Cart

Users will want to view their shopping cart contents, so let's implement this feature next. The cart we've implemented for demonstration purposes is pretty simple, consisting of a list of products currently residing in the cart, and a purchase form for completing the transaction, as depicted in the below screenshot:



Viewing the shopping cart

To implement this feature, begin by adding the following route to `app/Http/routes.php`:

```
1         Route::get('cart', 'CartController@index');
```

Next, add the following `index()` method to `CartController.php`:

```
1  public function index()
2  {
3      $cart = Auth::user()->cart;
4      return view('cart.index', compact('cart'));
5  }
```

We wish there were more to say about the method, but it really is that easy! All you need to do is return the authenticated user's cart and then pass it into the associated view. The view is similarly very easy to implement. We'll paste the relevant portion of `resources/views/cart/index.blade.php` here:

```
@if (count($cart) == 0)
    <p>Your cart is currently empty</p>
@else
    <table class="table table-border">
      <thead>
      <tr>
          <th></th>
          <th>Name</th>
          <th>Price</th>
      </tr>
      </thead>
      <tbody>
      @foreach ($cart as $item)
        <tr>
        <td><a href="/cart/remove/{{ $item->id }}">x</a></td>
        <td>{{ $item->product->name }}</td>
        <td>${{ $item->product->price }}</td>
        </tr>
      @endforeach
      </tbody>
    </table>

@endif
```

This is obviously a pretty simple implementation, intended to provide you with a starting point. For instance, one immediately desirable feature would be to total the product prices and provide the user with a total price.

Now that users can view their shopping cart, let's implement the product removal feature.

Removing Products from the Cart

In the shopping cart view (`resources/views/cart/index.blade.php`) code presented above, notice that each product line item is accompanied by a link which when clicked, removes the product from the cart:

```
        <td><a href="/cart/remove/{{ $item->id }}">x</a></td>
```

To implement this feature let's first add the following route to the `app/Http/routes.php` file:

```
        Route::get('cart/remove/{id}', 'CartController@remove');
```

Next, add the following `remove()` method to the `CartController` controller:

```
public function remove($id)
{
    Auth::user()->cart()
      ->where('id', $id)->firstOrFail()->delete();
    return redirect('/cart');
}
```

As with the `index()` action, this is very easy to implement. The `$id` is passed along via the URI, and subsequently used to retrieve the specific record in the user's shopping cart. Because we're using the `where` clause you'll additionally need to use `firstOrFail()` to retrieve just the data rather than have to treat it as an array. Once retrieved, the record is deleted, and the user is returned to the shopping cart view page.

Processing the Cart Order

With the cart addition, viewing, and removal features in place, all that remains is to provide a solution for ordering the cart products! If you have a look at the WeDewLawns.com's `resources/views/cart/index.blade.php` file you'll see we've included a simple purchase form containing fields for the user's billing and shipping information, credit card number, expiration date, and security code. When the `Complete Order` button is clicked, the user will be taken to the `/cart/complete` URI, so let's implement that URI's associated action now. Begin by adding the following route to your `app/Http/routes.php` file:

```
        Route::post('cart/complete', [
            'as' => 'cart.complete',
            'uses' => 'CartController@complete'
        ]);
```

With the new route in place, add the following `complete()` method to the `CartController`. We'll present the code first, followed by an explanation of key syntax:

```
        use App\Order;

        ...

public function complete(Request $request)
{
  $user = Auth::user();

  $total = $user->cart->sum(function($item){
      return $item->product->priceToCents();
```

```
    });

    $charge = $user->charge($total, [
        'source' => $request->get('stripe_token'),
        'receipt_email' => $user->email,
        'metadata' => [
            'name' => $user->name,
        ],
    ]);

    if (! $charge) {
        return back()->withErrors('Charge Failed');
    }

    // Add the order
    $order = new Order();
    $order->order_number = $charge->id;
    $order->email = $user->email;
    $order->billing_name = $request->input('card_name');
    $order->billing_address = $request->input('address');
    $order->billing_city = $request->input('city');
    $order->billing_state = $request->input('state');
    $order->billing_zip = $request->input('zip');
    $order->billing_country = $request->input('country');

    $order->shipping_name =
      $request->input('shipping_name');
    $order->shipping_address =
      $request->input('shipping_address');
    $order->shipping_city =
      $request->input('shipping_city');
    $order->shipping_state =
      $request->input('shipping_state');
    $order->shipping_zip =
      $request->input('shipping_zip');
    $order->shipping_country =
      $request->input('shipping_country');
    $order->save();

    // Update the old cart
    foreach ($user->cart as $cart) {
        $cart->order_id = $order->id;
```

```
53          $cart->complete = 1;
54          $cart->save();
55      }
56
57      return view('checkout.thankyou',
58        compact('order', 'charge'));
59  }
```

Let's review the code:

- First up we use Laravel's slick collection summation feature to total the price of all products found in the shopping cart.
- Next the customer's credit card is charged in the very same manner in which the charge was implemented in the previous chapter. If this looks like magic to you please refer back to the earlier discussion. If the charge attempt fails, the user is returned to the shopping cart with an error message.
- The billing and shipping information are added to the new `Order` object, and saved to the database.
- Next, each item in the cart is identified as "complete", meaning the order was successfully fulfilled.
- Finally, the customer is sent to the `checkout/thankyou.blade.php` view and provided with confirmation of successful purchase.

Where to From Here?

This initial chapter iteration is intended to demonstrate one particularly easy solution for integrating a shopping cart into your Laravel application. While it gets the job done, plenty of work remains. Here are a few feature ideas you might consider implementing in order to further improve and streamline the ordering and fulfillment process:

- Add a cart price total to the shopping cart view page
- Extend the administration console to provide the We Dew Lawns, Inc. fulfillment center with a summary of completed shopping cart orders.
- Give customers the ability to store their billing and shipping information in the user profile, and then prepopulate the purchase form.

The sky is the limit in terms of where you can take the store from here!

The Project Presentation

Clouds of choking cigar smoke swirl around the office. Mr. McDew's feet are propped up on the desk, and he's reading the latest copy of "Landscaper Luxuries", a new magazine catering to the nouveau riche of the landscaping industry.

TM: Well, well, well, it looks like Mr. Pocket Calculator has finally arrived. How much geometry do you use on my website, son?

MPC (that's you): Geometry, sir? I don't think we use any.

TM: That's good to hear. Never trusted that stuff anyway, too many angles. Anyway, are customers able to use a shopping cart now?

MPC: Yes sir. Customers can now add and remove products from their shopping cart, as well as view their cart and purchase their cart contents.

TM: Fantastic work son. Can I interest you in a cigar?

MPC: No thank you sir, I don't smoke.

TM: Well, it's never too late to start, so I'll light this one up for you. Now let me tell you about a new website feature I've been thinking about…

Summary

Congratulations, you've successfully implemented a complete online store offering one-time and shopping cart-based purchases, and subscription capabilities. While this chapter currently represents the book's conclusion, we're always working on new material and so if you have any ideas be sure to contact us at support@easyecommercebook.com!