O'REILLY®

Laravel Up & Running

A FRAMEWORK FOR BUILDING MODERN PHP APPS

Matt Stauffer

Laravel: Up and Running

Matt Stauffer

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Laravel: Up and Running
by Matthew E. Stauffer

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

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

Acquisitions Editor: FILL IN
Editor: TO COME
Production Editor: TO COME
Copyeditor: TO COME
Proofreader: TO COME
Indexer: TO COME
Interior Designer: David Futato
Cover Designer: TO COME
Illustrator: Rebecca Demarest

Month Year: First Edition

Revision History for the First Edition
2016-03-17: First Early Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449370787* for release details.

xxx-x-xxx-xxxx-x

[???]

Table of Contents

CHAPTER 1

Why Laravel?

In the early days of the dynamic web, programming a website or application meant writing the code for not just the unique business logic of your application, but also for each of the components that are so common across sites—user authentication, input validation, database access, templating, and more.

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

So, why frameworks? And, more specifically, why Laravel?

Why use a framework?

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

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

“I’ll just build it myself”

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

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

Consistency + Flexibility

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

When someone prescribes rolling your own framework for each new project, what they’re really advocating is the ability to *control* what does and doesn’t go into your application’s foundation. That means the best frameworks will both provide you with a solid foundation, but also give you the freedom to customize to your heart’s content.

A short history of web and PHP frameworks

We’ve covered why frameworks are helpful, and that’s an important aspect of understanding the answer to the question “Why Laravel?” But it’s also valuable to know what frameworks and other movements happened in the PHP and web development spaces prior to Laravel’s rise to popularity.

Ruby on Rails

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

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

The influx of PHP frameworks

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

CakePHP was the first in 2005, and it was soon followed by Symfony, CodeIgniter, Zend Framework, and Kohana (a CodeIgniter fork). Yii arrived in 2008, and Aura and Slim in 2010. 2011 brought Fuel, another CodeIgniter fork, and Laravel, which wasn't quite a CodeIgniter offshoot, but instead proposed as an alternative.

Some of these frameworks were more Rails-y, focusing on database ORMs, MVC structures, and other tools targeting rapid development. Others, like Symfony and Zend, focused more on enterprise design patterns and e-commerce.

The good and the bad of CodeIgniter

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

CodeIgniter was simple, easy to use, and boasted amazing documentation and a strong community. But it grew slowly, and as the framework world grew and PHP's tooling advanced, CodeIgniter started falling behind both in terms of technological advances and out-of-the-box features. It was in 2010 that Taylor Otwell, Laravel's creator, became dissatisfied enough with CodeIgniter that he set off to write his own framework.

@todo Cite this: http://maxoffsky.com/code-blog/history-of-laravel-php-framework-eloquence-emerging/ Attribute more core folks from 1? 5? TAYLOR: 1.0 - 2.0 - xikeon, mikelbring, pedroborges jason lewis... 3.0 - dayle, phill sparks, machuga, jason lewis.. 4.0 -5.0 - transitioning from 3.0 crew to matt, adam, jeffrey

Laravel 1, 2, and 3

The first beta of Laravel 1 was released in June 2011, and it consisted of entirely custom code. It featured a custom ORM (Eloquent), Closure routing (inspired by Ruby Sinatra), a module system for extension, and helpers for forms, validation, authentication, and more.

Early Laravel development moved quickly, and Laravel 2 and 3 were released in November 2011 and February 2012 respectively. They introduced controllers, unit testing, a CLI tool, an IOC container, Eloquent relationships, and migrations.

Laravel 4

With Laravel 4, Taylor re-wrote the entire framework from the ground up. By this point Composer was showing signs of becoming an industry standard and Taylor saw the value of re-writing the framework as a collection of components, distributed and bunded together with Composer.

Taylor developed a set of components under the code-name Illuminate and, in May of 2013, released Laravel 4 as a fresh look at Laravel, based on pulling in Symfony and Illuminate packages in via Composer.

Laravel 4 also introduced queues, a mail component, Façades, and database seeding. And because Laravel was now relying on Symfony components, it was announced that Laravel would be mirroring (not exactly, but soon-after) the release-every-6-months release schedule Symfony follows.

Laravel 5

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

Laravel 5 introduced a revamped directory structure, removal of the form and HTML helpers, the introduction of the Contract interfaces, a spate of new views, Socialite for social media authentication, Elixir for asset compilation, Scheduler to simplify cron, dotenv for simplified environment management, Form Requests, and a brand new CLI.

The philosophy of Laravel

You only need to read through the Laravel marketing materials and READMEs to quickly gather its values. "Illuminate." "Spark." And then there are these: "Artisans." "Elegant." Also, these: "Breath of Fresh Air." "Fresh start." And finally: "Rapid." "Warp Speed."

The two most strongly communicated values of the framework are to increase developer speed and happiness. Taylor has described the Artisan language as intentionally contrasting against more utilitarian values. (@todo introduce a quote from Taylor interviewing where eh talks about making beautiful and elegant code that you really care about). And he's often talked about the value of making it easier and quicker for developers to take their ideas to fruitiion, getting rid of unnecessary barriers to creating great products.

Developer happiness

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

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

The Laravel Community

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

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

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

What makes Laravel unique?

@todo Ally's note: Maybe pulling out a few highlights of what separates Laravel from other frameworks in the space. You've already called out community and emphasis on the individual developer, anything else?

See how it works

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

Let's look at **Hello, World**.

Example 1-1. "Hello, World" in routes.php

```
// File: app/Http/routes.php
<?php

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

If you initialize a brand new Laravel application on your machine, edit the `app/Http/routes.php` file and make it look like the preceding example, and then serve the site from the `public` directory, you'll have a fully functioning **Hello, World** example:

Figure 1-1. New Laravel landing page

It looks very similar to do the same with controllers:

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

```
// File: app/Http/routes.php
<?php

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

// File: app/Http/Controllers/WelcomeController.php
<?php
namespace app\Http\Controllers;

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

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

Example 1-3. Multi-greeting "Hello, World" with database access

```
// File: app/Http/routes.php
<?php

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

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

use Illuminate\Database\Eloquent\Model;

class Bio extends Model
{
    protected $table = 'greetings';
}

// File: database/migrations/2015_07_19_010000_create_greetings_table.php
<?php

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

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

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

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

Why Laravel?

So—why Laravel?

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

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

CHAPTER 2

Setting Up a Laravel Development Environment

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

System Requirements

All of the following is possible with Windows systems, but many pages of instructions and caveats need to be made for Windows systems. As such, I'll leave those caveats to better-equipped writers online, and just focus on Unix/Linux/OS X developers.

Even with access to the command line and with the ability to install PHP and MySQL and other tools locally, you'll likely still run into version mismatches at some point or another, and it's highly recommended to do all of your local development on virtual machines using a tool like Vagrant.

Regardless of what tool you use, here are the minimum requirements for running Laravel 5.1:

- PHP >= 5.5.9
- OpenSSL PHP Extension
- PDO PHP Extension
- Mbstring PHP Extension
- Tokenizer PHP Extension

Tools

Composer

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

Vagrant, VMWare, and VirtualBox

If you're not familiar with Vagrant, it's a configuration tool that sits on top of either VMWare or VirtualBox and makes it easy to spin up virtual machines with predefined configurations. This means you can develop web sites locally without having to even run a web server on your local machine, and you can ensure your server configuration is in close sync with your production environment.

Laravel Homestead

Laravel Homestead is another tool that sits on top of Vagrant and provides a preconfigured virtual machine image that is perfectly set-up for Laravel development, *and* mirrors the most common VPS server that many Laravel sites run on.

Setting up Homestead

If you're new to Laravel development, getting started with VirtualBox, Vagrant, and Homestead will give you the best development experience regardless of your own computer's configuration.

What tools does Homestead offer?

You can always upgrade your Homestead box, but here's what it comes with by default:

- Ubuntu
- PHP
- Nginx
- MySQL
- Postgres
- Redis
- Memcached

- Node
- Beanstalkd

Installing Homestead's dependencies

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

Next, download and install Vagrant.

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

Installing Homestead

Next, let's actually install Homestead. You can install multiple instances of Homestead (often used to host a different Homestead box per project), but I prefer a single Homestead virtual machine for all of my projects. If you want one per project, you'll want to install Homestead in your project directory; check the Homestead documentation online for instructions. If you want a single virtual machine for all of your projects, install Homestead in your user's home directory like in Example 2-1.

Example 2-1. Installing Homestead

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

Now, run the initialization script from wherever you put the `Homestead` directory like in Example 2-2.

Example 2-2. Initializing Homestead

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

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

Configuring Homestead

Open up `Homestead.yaml` and configure it how you'd like. You'll need to tell it your provider (likely `virtualbox`), point it to your public SSH key (likely `~/.ssh/id_rsa.pub`), map folders and sites to their local machine equivalents, and provision database.

Mapping folders in Homestead allows you to edit files on your local machine and have those files show up in your Vagrant box so they can be served. For example, if you have a `~/Sites` directory where you put all of your code, you would map the folders in Homestead like in Example 2-3.

Example 2-3. Mapping folders in Homestead.yaml

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

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

TLDs for development sites

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

Now, let's set up our first example web site. Let's say our live site is going to be `projectName.com`. Let's map our local development folder to `projectName.app`, so we have a separate URL to visit for local development.

Example 2-4. Mapping sites in Homestead.yaml

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

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

Finally, we're going to need to teach your local machine that, when you try to visit `projectName.app`, it should look at your computer's local IP Address to resolve it. Mac and Linux users should edit `/etc/hosts`, Windows users `C:\Windows\System32\drivers\etc\hosts`. We'll just add a line to this file that looks like Example 2-5.

Example 2-5. Adding a local development site to your hosts file

```
192.168.10.10   projectName.app
```

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

Creating databases in Homestead

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

Example 2-6. Creating databases in Homestead.yaml

```
databases:
    - projectname
```

Provisioning Homestead

Since this is our first time actually turning on our Homestead box, we need to tell Vagrant to initialize it. Navigate to your Homestead directory and run `vagrant up`:

Example 2-7. Provisioning a Homestead box

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

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

Using Homestead day-to-day

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

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

- `vagrant up` spins up the Homestead box
- `vagrant suspend` takes a snapshot of where the box is and then shuts it down; like "hibernating" a desktop machine
- `vagrant halt` shuts the entire box down; like turning off a desktop machine

- `vagrant destroy` deletes the entire box; like formatting a desktop machine
- `vagrant provision` re-runs the provisioners on the preexisting box

Connecting to Homestead databases from desktop applications

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

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

Creating a new Laravel project

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

You can learn about both options in more detail at the Installation Documentation: *http://laravel.com/docs/installation*

Installing Laravel with the Laravel installer tool

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

Example 2-8. Installing the Laravel installer tool

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

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

Example 2-9. Creating a new Laravel project using the installer tool

```
laravel new projectName
```

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

Installing Laravel with Composer's `create-project` feature

Composer also offers a feature called `create-project` for creating new projects with a particular skeleton. To use this tool to create a new Laravel project, issue the command shown in Example 2-10.

Example 2-10. Creating a new Laravel project using the installer tool

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

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

Laravel's Directory structure

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

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

Let's walk through them one-by-one to get familiar.

The loose files

`.env` and `.env.example` are the files that dictate the environment variables, variables which are expected to be different in each environment and are therefore not committed to version control. `.env.example` is a template that each environment should duplicate to create its own `.env` file, which is Git ignored.

`artisan` is the file that allows you to run Artisan commands from the command line.

`.gitignore` and `.gitattributes` are Git configuration files.

`composer.json` and `composer.lock` are the configuration files for Composer; `composer.json` is user-editable and `composer.lock` is not. These files share some basic information about this project and also define its PHP dependencies.

`gulpfile.js` is the (optional) configuration file for Elixir and gulp. This is for managing your frontend assets.

`package.json` is like `composer.json` but for frontend assets.

`phpspec.yml` and `phpunit.xml` are configuration files for testing tools.

`readme.md` is a Markdown file giving a basic introduction to Laravel.

`server.php` is a backup server that tries to allow less-capable servers to still preview the Laravel application.

The folders

`app` is where the bulk of your actual application will go. Models, controllers, route definitions, commands, and your PHP domain code all go in here.

`bootstrap` contains the files that the Laravel framework uses to boot every time it runs.

`config` is where all configuration files live.

`database` is where database migrations and seeds live.

`public` is the directory the server points to when it's serving the website. This contains `index.php`, which is the front controller that kicks off the bootstrapping process and routes all requests appropriately. It's also where any public-facing files like images, stylesheets, scripts, or downloads go.

`resources` is where non-PHP files that are needed for other scripts live. Views, language files, and (optionally) Sass/LESS and source JavaScript files live here.

`storage` is where caches, logs, and compiled system files live.

`tests` is where unit and integration tests live.

`vendor` is where Composer installs its dependencies. It's Git ignored, as Composer is expected to run as a part of your deploy process on the remote server.

Up and Running

You're now up and running with a bare Laravel install. Run `git init`, commit the bare files, and you're ready to start coding.

Testing

In every chapter after this, the "Testing" section of the *Testing & TL;DR* conclusion to each chapter will show you how to write tests for that feature. Since this chapter doesn't cover a testable feature, let's talk tests quickly. To learn more about writing and running tests in Laravel, head over to ???.

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

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

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

TL;DR

Laravel has a pre-configured Vagrant setup named Homestead, which is the recommended local development environment. Laravel relies on, and can be installed by, Composer, and comes out of the box with a series of folders and files that reflect both its conventions and its relationship with other open source tools.

CHAPTER 3

Routing and Controllers

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

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

Route Definitions

Laravel's routes are defined in `app/Http/routes.php`.

The simplest route definition matches a URI (e.g. `/`) with a Closure:

Example 3-1. Basic Route Definition

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

What's a Closure?

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

This teaches the Laravel router that, if anyone visits `/` (the root of your domain), it should run the Closure defined there and return the result. Note that we `return` our content, not `echo` or `print`.

A Quick Introduction to Middleware

You might be wondering, "Why am I returning *Hello, World!* instead of echoing it?"

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

Many simple web sites could be defined entirely within the Routes file. Simple `GET` routes combined with a few templates can serve a classic web site easily.

Example 3-2. Sample web site

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

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

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

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

Static calls

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

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

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

HTTP Methods

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

Route verbs

You might've noticed the "`get`" method in `Route::get`. This means we're telling Laravel to only match for this route when the HTTP request uses the `GET` action. But what if it's a form `POST`, or maybe some JavaScript sending `PUT` or `DELETE` requests? There are a few other options for methods to call on a route definition.

Example 3-3. Route verbs

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

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

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

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

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

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

Route Handling

As you've probably guessed, passing a Closure to the route definition is not the only way to teach it how to resolve a route. Closures are quick and simple, but the larger your application gets, the clumsier it becomes to put all of your routing logic in this one file.

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

Example 3-4. Routes Calling Controller Methods

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

This is telling Laravel to pass requests to that URI to `App\Http\Controllers\WelcomeController`, `welcome()` method. This method will be passed the same parameters and treated the same as a Closure you might've alternately put in its place.

Route Parameters

If the route you're defining has parameters, it's simple to add them into both the URI definition and the callback.

Example 3-5. Route Parameters

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

The naming relationship between route parameters and Closure/controller method parameters

As you can see in Example 3-5, it's most common to use the same name for your route parameters (`{id}`) and the method parameters they inject into your route definition (`$id`+). But is this necessary?

Unless you're using route/model binding, no. In fact, you could dependency inject to the left and right of a method parameter *and* you could name it different, and it would still work. The only thing that defines which route parameter matches with which method parameter is that they are in the same order (left to right), excluding injected dependencies, as you can see below.

```
Route::get('users/{id}', function (
    Application $injectedApplication,
    $thisIsActuallyTheRouteId,
    Request $injectedRequest
    ) {
        //
    });
```

You can also make your route parameters optional:

Example 3-6. Optional Route Parameters

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

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

Example 3-7. Regular Expression Route Contraints

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

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

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

As you've probably guessed, if you visit a URI that matches a path, but the regex doesn't match the parameter, it won't be matched. So `users/abc` in the example above would skip the first Closure, but it would be matched by the second Closure, so it would get routed there. On the other hand, `posts/abc/123` wouldn't match any of the Closures, so it would give a `404`.

Route Names

By default, you'll refer to these routes elsewhere in your application just by their URI. This will be very familiar. There's a `url()` helper to simplify that linking in your views, if you need it; see Example 3-8 for an example.

Example 3-8. URL Helper

```
<a href="<?php echo url('/'); ?>">
```

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

Example 3-9. Defining route names

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

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

We've introduced a few new concepts there. First, we passed a configuration array to the second parameter instead of a string. Laravel checks the type of the second parameter and routes accordingly; if it's a Closure, it runs it; if it's a string, it assumes it's the identification of a controller and method; and if it's an array, it expects to get enough parameters frmo the array that it can resolve the route.

In Example 3-9, we can see that we've also introduced the idea of `as`, which allows us to name the route. We've named this route `members.show`, which is a common convention within Laravel for route and view names: `resourcePlural.action`.

Route naming conventions

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

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

To learn more about these conventions, read about "Resource controllers" on page 40.

And finally, we showed that the configuration array syntax should have a property with the key `uses` if it's going to refer to a controller method. If you want to name a Closure route, just pass the Closure in with no key like in Example 3-10.

Example 3-10. Defining Closure routes with a configuration array

```
Route::get('/members/{id}/edit', [
    'as' => 'members.edit',
    function ($id) {
        //
    }
]);
```

We also introduced the `route()` helper. Just like `url()`, it's intended to be used in views to simplify linking to a named route. If the route has no parameters, you can simply pass the route name: `route(`*`members.index`*`)`. If it has parameters, pass them in as an array as the second parameter like we did in example Example 3-9.

Passing route parameters to the `route()` helper

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

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

OPTION 1.

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

OPTION 2.

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

OPTION 3.

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

OPTION 4.

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

As you can see, non-keyed array values are assigned in order; keyed array values are matched with the route parameters matching their key; and anything left over is added as a query parameter.

Route Groups

Often a group of routes share a particular characteristic—a certain authentication requirement, a URI prefix, or maybe a controller namespace.

Route groups allow you to group several routes together in one to reduce duplication of the route definitions and to increase the clarity of the various segments of the routes file.

To group two or more routes together, you "surround" the route definitions with a group; in reality, you're actually passing a Closure to the group definition, and defining the grouped routes within that Closure.

Example 3-11. Defining a route group

```
Route::group([], function () {
    Route::get('hello', function () {
        return 'Hello';
    });
    Route::get('world', function () {
        return 'World';
    });
});
```

By default, a route group doesn't actually do anything. There's no difference between the group in Example 3-11 and separating a segment of your routes with code comments.

The empty array that's the first parameter, however, allows you to pass a variety of configuration settings that will apply to the entire route group.

Route group middleware

Probably the most common use for route groups is to apply middleware to a group of routes. We'll learn more about middleware in chapter ???, but, among other things, they're what Laravel uses for authenticating users and restricting guest users from using certain parts of a site.

Example 3-12. Restricting a group of routes to logged-in users only

```
Route::group(['middleware' => 'auth'], function () {
    Route::get('dashboard', function () {
        return view('dashboard');
    });
    Route::get('account', function () {
        return view('account');
    });
});
```

Route group route prefix

If you have a group of routes that share a segment of their route—for exxample, if your site's API is prefixed with `api`—you can use route groups to simplify this structure.

Example 3-13. Prefixing a route with route groups

```
Route::group(['prefix' => 'api'], function () {
    Route::get('/', function () {
        //
    });
    Route::get('users', function () {
        //
    });
});
```

Note that each prefixed group also has a `/` group that represents the root of the prefix —in Example 3-13 that's `/api`.

Route group sub-domain routing

Sub-domain routing is the same as route prefixing, but it's scoped by subdomain instead of route prefix. There are two primary uses for this. First, to present different sections of the application (or entirely differently applications) to different subdomains:

Example 3-14. Sub-domain routing

```
Route::group(['domain' => 'api.myapp.com'], function () {
    Route::get('/', function () {
        //
    });
});
```

And second, to set part of the subdomain as a parameter—most often used in cases of multitenancy (think Slack or Harvest, where each company gets their own subdomain like `tighten.slack.co`).

Example 3-15. Parameterized sub-domain routing

```
Route::group(['domain' => '{account}.myapp.com'], function () {
    Route::get('/', function ($account) {
        //
    });
    Route::get('users/{id}', function ($account, $id) {
        //
    });
});
```

Note that any parameters for the group get passed into the grouped routes' methods as the first parameter(s).

Route group namespace prefix

When routes are grouped by subdomain or route prefix, it's likely their controllers have a similar namespace. In the API example, all of the API routes' controllers might be under an API namespace.

Example 3-16. Route group namespace prefixes

```
// App\Http\Controllers\ControllerA
Route::get('/', 'ControllerA@index');

Route::group(['namespace' => 'API'], function () {
    // App\Http\Controllers\API\ControllerB
    Route::get('/', 'ControllerB@index');
});
```

Route group name prefix

The prefixes don't stop there. It's common that route names will reflect the inheritance chain of URI elements, so `users/comments/5` will be served by a route named `users.comments.show`. In this case, it's common to use a route group around all of the routes that are beneath the `users.comments` resource. Using the name prefix on a route group can make this simpler.

Example 3-17. Route group name prefixes

```
Route::group(['as' => 'users.', 'prefix' => 'users'], function () {
    Route::group(['as' => 'comments.', 'prefix' => 'comments'], function () {
        // Route name will be users.comments.show
        Route::get('{id}', ['as' => 'show', function () {
            //
        }]);
    });
});
```

Views

In a few of the route Closures we've looked at so far, we've seen `return view(`*`account`*`)` or something similar. What's going on here?

If you're not familiar with the MVC pattern, Views (or Templates) are files that describe how some particular output should look like. You might have views for JSON or XML or emails, but the most common views in a web framework output HTML.

In Laravel, there are two formats of view you can use out of the box: Blade (see Chapter 4) or PHP. The difference is in the filename: `about.php` will be rendered with the PHP engine, and `about.blade.php` will be rendered with the Blade engine.

Three ways to view()

There are also three different ways to do `view()`. For now, just concern yourself with `view()`, but if you ever see `View::make()`, it's the same thing, and you could also inject the `Illuminate\View\ViewFactory` if you prefer.

So, once you've imported a view, you have the option to simply return it, which will work fine if the view doesn't rely on any variables from the controller.

Example 3-18. Simple view() usage

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

This code in Example 3-18 looks for a view in `resources/views/home.blade.php` or `resources/views/home.php`, and loads its contents and parses any inline PHP or control structures until you have just the view's output. Once you return it, it's passed on to the rest of the application and eventually returned to the user.

But what if you need to pass in variables?

Example 3-19. Passing variables to views

```
Route::get('tasks', function () {
    return view('tasks.index')
        ->with('tasks', Task::all());
});
```

This Closure loads the `resources/views/tasks/index.blade.php` or `resources/views/tasks/index.php` view and passes it a single variable named `tasks`, which contains the result of the `Task::all()` method, which is a database query we'll learn about in ???.

If you prefer non-fluent routing, you could pass an array of variables as the second parameter:

Example 3-20. Passing variables to views in an array

```
Route::get('tasks', function () {
  return view('tasks.index', ['tasks' => Task::all()]);
});
```

View Composers and sharing variables with every view

Sometimes it can become a hassle to pass the same variables over and over. There may be a variable that you want accessible to every view in the site, or to a certain class of views or a certain included sub-view—for example, all views related to tasks, or the header partial.

It's possible to share certain variables with every template or just certain templates, like in the following code:

Example 3-21.

```
view()->share('variableName', 'variableValue');
```

To learn more, check out the View composers and service injection section in Chapter 4.

Controllers

@todo: Add a graphic to explain MVC

I've mentioned controllers a few times, but until now most of the examples have shown route Closures. If you're not familiar with the MVC pattern, Controllers are essentially classes that organize the logic of one or more routes together in one place. Controllers tend to group similar routes together, especially if your application is structured along a traditionally CRUD-like format; in this case, a controller might handle all the actions that can be performed on a particular resource.

It may be tempting to cram all of the application's logic into the controllers, but it's better to think of controllers as the traffic cop that routes HTTP requests around your application. Since there are other ways requests can come into your application —cron jobs, "Artisan" command line calls, queue jobs, etc.--it's wise to not rely on controllers for much behavior. This means a controller's primary job is to capture the intent of an HTTP Request and pass it onto the rest of the application.

So, let's create a controller. There's an artisan command for that, so from the command line run this command:

```
php artisan make:controller MySampleController
```

Artisan and Artisan Generators

Laravel comes bundled with a command-line tool called Artisan. Artisan can be used to run migrations, create users and other database records manually, and many other manual, one-time tasks.

Under the `make` namespace, Artisan provides tools for generating skeleton files for a variety of system files. That's what allows us to run `php artisan make:controller`.

To learn more about this and other Artisan features, see ???.

This will create a new file named `MySampleController.php` in `app/Http/Controllers`.

Example 3-22. Default generated controller

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

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

class MySampleController extends Controller
{
    // A lot of methods
}
```

For now, just delete all those methods—we'll talk about them in a second. Create a new public method called *home()* and we'll just return some text there.

Example 3-23. Simplest controller example

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;

class MySampleController extends Controller
{
    public function home()
    {
        return 'Hello, World!';
    }
}
```

And, like we learned before, we'll hook up a route to it:

Example 3-24. Route for the simplest controller

```
// app/Http/routes.php
<?php

Route::get('/', 'MySampleController@home');
```

That's it. Visit the / route and you'll see the words "Hello, World!".

Controller Namespacing

In Example 3-24 we referenced a controller with the fully-qualified classname of `App\Http\Controllers\MySampleController`, but we only used the class name. This isn't because we can simply reference controllers by their class name. Rather, we can ignore `App\Http\Controllers\` when we reference controllers.

This means that if you have a controller with the fully-qualified class name of `App\Http\Controllers\API\ExercisesController`, you'd reference it in a route definition as `API\ExercisesController`.

The most common use of a controller method, then, will be something like this:

Example 3-25. Common controller method example

```
// TasksController.php
...
public function index()
{
    return view('tasks.index')
        ->with('tasks', Task::all());
}
```

This controller method loads the `resources/views/tasks/index.blade.php` or `resources/views/tasks/index.php` view and passes it a single variable named `tasks`, which contains the result of the `Task::all()` Eloquent method.

Getting user input

The second most common action to perform in a controller method, however, is to take input from the user and act on it. That introduces a few new concepts, so let's take a look at a bit of sample code and walk through the new pieces.

First, let's bind it quickly; see Example 3-26.

Example 3-26. Binding basic form actions

```
// app/Http/routes.php
Route::get('tasks/create', 'TasksController@create');
Route::post('tasks', 'TasksController@store');
```

Notice that we're binding the `get` route of `resources/create` (which shows the form) and the `post` route of `resources/` (which is where we post when we're creating a new resource). We can assume the `create` method in our controller just shows a form, so let's look at the `store` method in Example 3-27.

Example 3-27. Common form input controller method

```
// TasksController.php
...
public function store()
{
    $task = new Task;
    $task->title = Input::get('title');
    $task->description = Input::get('description');
    $task->save();

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

We're seeing Eloquent models and the `redirect()` functionality, and we'll talk about them more later, but you can see what we did here: Create a new `Task`, pull data out of the user input and set it on the task, save it, and then redirect back to the all tasks page.

There are two main ways to get user input from a `POST`: the `Input` Façade, which we used above, and the `Request` object, which we'll talk about next.

As you can see, we can get the value of any user-provided information, whether from a query parameter or a `POST` value, using `Input::get(`*`fieldName`*`)`. So our user filled out two fields on the "add task" page: "Input" and "Description." We get both using the `Input` Façade, save it to the database, and then return.

Injected Dependencies into Controllers

Laravel's Façades present a simple interface to the most useful classes in Laravel's codebase. You can get information about the current request and user input, the session, caches, and much more.

But if you prefer to inject your dependencies, or if you want to use a service that doesn't have a Façade, you'll need to find some way to get your dependencies in.

This is our first exposure to Laravel's Service Container, so if you want to know more about how this is actually functioning, check out chapter ??? to learn more. But for now, you can think about it as a little bit of Laravel Magic.

All controller methods (including the constructors) are resolved out of Laravel's Container, which means anything you typehint that the Container knows how to resolve will be automatically injected.

As a nice example, what if you'd prefer having an instance of the `Request` object instead of using the Façade? Just typehint `Illuminate\Http\Request` in your method parameters like in Example 3-28.

Example 3-28. Controller method injection via typehinting

```
// TasksController.php
...
public function store(\Illuminate\Http\Request $request)
{
    $task = new Task;
    $task->title = $request->input('title');
    $task->description = $request->input('description');
    $task->save();

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

So, you've defined a parameter that must be passed into this method. And since you typehinted it, and since Laravel knows how to resolve that class name, you're going to have the `$request` object ready for you to use in your method with no work on your part. No explicit binding, no anything else—it's just there.

Resource controllers

Sometimes naming the methods in your controllers can be the hardest part of writing a controller. Thankfully, Laravel has some conventions for all of the routes of a traditional REST/CRUD controller (called a "Resource controller" in laravel); additionally, it comes with a generator out of the box and a convenience route definition that allows you to bind an entire resource controller at once.

To see the methods that Laravel expects for a resource controller, let's generate a new controller from the command line:

```
php artisan make:controller MySampleResourceController
```

Now open `app/Http/Controllers/MySampleResourceController.php`. You'll see it comes pre-filled with quite a few methods. Let's walk over what each represents. We'll use a `Task` as an example.

The methods of Laravel's resource controllers

For each, you can see the HTTP Verb, the URL, the controller method name, and the "name":

Verb	URL	Controller method	Name	Description
GET	/tasks	index	tasks.index	Show all tasks
GET	/tasks/create	create	tasks.create	Show the create task form
POST	/tasks	store	tasks.store	Accept form submission from the create task form
GET	/tasks/{photo}	show	tasks.show	Show one task
GET	/tasks/{photo}/edit	edit	tasks.edit	Edit one task
PUT/PATCH	/tasks/{photo}	update	tasks.update	Accept form submission from the edit task form
DELETE	/tasks/{photo}	destroy	tasks.destroy	Delete one task

Binding a resource controller

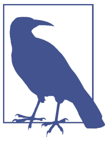
So, we've seen both that these are the conventional route names to use in Laravel, and also that it's easy to generate a resource controller with all the route names. Thankfully, you don't have to bind all those routes every time if you don't want. Instead, there's a trick for that—take a look at Example 3-29.

Example 3-29. Resource controller binding

```
// app/Http/routes.php
Route::resource('tasks', 'TasksController');
```

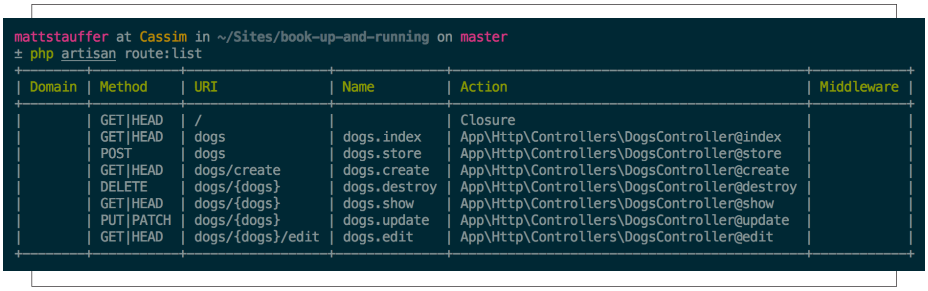
Not only will this automatically bind all of the resource routes for you, but it'll also create a pre-defined set of route names for all of your auto-bound routes. That means, for example, the `index` method on the `tasks` resource controller will get named `tasks.index`.

As you can probably guess, these are both useful as convenience tools for binding an entire resource at once, but also just as guidelines for how to go about naming your own methods.



Artisan route:list

If you ever find yourself in a situation wondering what routes your current application has available, there's a tool for that: From the command line, run `php artisan route:list` and you'll get a listing of all of the available routes.



```
mattstauffer at Cassim in ~/Sites/book-up-and-running on master
± php artisan route:list
+--------+-----------+------------------+--------------+---------------------------------------------+------------+
| Domain | Method    | URI              | Name         | Action                                      | Middleware |
+--------+-----------+------------------+--------------+---------------------------------------------+------------+
|        | GET|HEAD  | /                |              | Closure                                     |            |
|        | GET|HEAD  | dogs             | dogs.index   | App\Http\Controllers\DogsController@index   |            |
|        | POST      | dogs             | dogs.store   | App\Http\Controllers\DogsController@store   |            |
|        | GET|HEAD  | dogs/create      | dogs.create  | App\Http\Controllers\DogsController@create  |            |
|        | DELETE    | dogs/{dogs}      | dogs.destroy | App\Http\Controllers\DogsController@destroy |            |
|        | GET|HEAD  | dogs/{dogs}      | dogs.show    | App\Http\Controllers\DogsController@show    |            |
|        | PUT|PATCH | dogs/{dogs}      | dogs.update  | App\Http\Controllers\DogsController@update  |            |
|        | GET|HEAD  | dogs/{dogs}/edit | dogs.edit    | App\Http\Controllers\DogsController@edit    |            |
+--------+-----------+------------------+--------------+---------------------------------------------+------------+
```

Figure 3-1. `php artisan route:list` example

Route model binding

One of the most common actions, particularly in a REST-style URL structure (e.g. `/conference/{conference_id}`) is that the first line of any controller method tries to find the resource with the given idea, like in Example 3-30.

Example 3-30. Getting a resource for each route

```
Route::get('conferences/{id}', function ($id) {
    $conference = Conference::findOrFail($id);
});
```

This is such a common behavior that Laravel has introduced a feature to simplify it called "Route Model Binding." This allows you to define that a particular parameter name (e.g. `{conference}`) will indicate to the route resolver that it should look up an Eloquent record with that ID and then pass it in as the parameter *instead* of the ID.

There are two kinds of route model binding: implicit and custom (or explicit).

Implicit route model binding

The simplest way to use route model binding is to name your route parameter something unique to that model (e.g. name it `$conference` instead of `$id`), then typehint that parameter in the Closure/controller method and use the same variable name there. It's easier to show than to describe, so take a look at Example 3-31.

Example 3-31. Using an explicit Route Model Binding

```
Route::get('conferences/{conference}', function (Conference $conference) {
    return view('conferences.show')->with('conference', $conference);
});
```

Because the route parameter (`{conference}`) is the same as the method parameter (`$conference`), and the method parameter is type-hinted with a `Conference` model (`Conference $conference`), Laravel sees this as a route model binding. Every time this rout is visited, the applicaiton will assume that whatever is passed into the URL in place of `{conference}` is an ID that should be used to look up a `Conference`, and then that resulting model instance will be passed in to your Closure or controller method.

Customizing the royte key for an Eloquent model

Any time an Eloquent model is looked up via a URL segment (usually because of route model binding), the default column Eloquent will look it up by is its primary key (ID).

To change the column your Eloquent model uses as its URL lookup, add a method to your model named `getRouteKeyName`:

```
    public function getRouteKeyName()
    {
        return 'slug';
    }
```

Now, a URL like `conferences/{conference}` will expect to get the slug instead of the ID, and will perform its lookups accordingly.

Custom route model binding

To manually configure Route Model bindings, go to the `boot()` method of `App\Providers\RouteServiceProvider` and add a line like in <<EX61>.

Example 3-32. Adding a Route Model Binding

```
    public function boot(Router $router)
    {
        parent::boot($router);

        $router->model('event', Conference::class);
    }
```

You've now defined that whenever a route has a parameter in its definition named `{event}`, the route resolve will return an instance of the `Conference` class with the ID of that URL parameter.

Example 3-33. Using an explicit Route Model Binding

```
Route::get('events/{event}', function (Conference $event) {
    return view('events.show')->with('event', $event);
});
```

Form method spoofing & CSRF

Sometimes, you need to pass information along to Laravel's router manually. CSRF tokens prove that the requesting form is actually coming from the same application, and have to be passed manually. And HTML forms only allow for `GET` or `POST`, so if you want any other sort of verb, you'll need to specify that yourself. Let's take a look at these two.

What is CSRF?

CSRF, or Cross-Site Request Forgery, is when one web site pretends to be another. The goal is for someone to hijack your users' access to your web site by submitting forms from *their* web site toward *your* web site, in the user's browser, while they're still logged into your site.

The best way around CSRF is to protect all inbound routes — `POST`, `DELETE`, etc.--with a token, which Laravel does out of the box.

An introduction to HTTP verbs

We've talked about the `GET` and `POST` HTTP verbs already. If you're not familiar with HTTP verbs, the other two most common are `PUT` and `DELETE`, but there's also `HEAD`, `OPTIONS`, `PATCH`, and two others that are pretty much never used in normal web development, `TRACE` and `CONNECT`.

Here's the quick rundown: `GET` requests a resource and `HEAD` asks for a headers-only version of the `GET`, `POST` creates a resource, `PUT` overwrites a resources and `PATCH` modifies a resource, `DELETE` deletes a resource, and `OPTIONS` asks the server which verbs are allowed at this URL.

HTTP verbs in Laravel

So, as we've shown already, you can define which verbs a route will match in the route definition, with the difference between `Route::get`, `Route::post`, `Route::any`, or `Route::match`.

But how does one send a request other than `GET` with a web browser? First, the method in an HTML form determines its HTTP verb: if your form has a `method` of "get", it will submit via query parameters and a GET method; if the form has a method of "post", it will submit via the post body and a POST method.

JavaScript frameworks make it easy to send other requests like DELETE and PATCH. But if you find yourself needing to submit forms in Laravel with verbs other than GET or POST, you'll need to use "form method spoofing".

Form method spoofing

To inform Laravel that the form you're currently submitting should be treated as something other than POST, add a hidden variable named _method with the value of either PUT, PATCH, or DELETE, and Laravel will match and route that form submission _as if it were actually a request with that verb._

Example 3-34. Form method spoofing

```
<form action="/tasks/5" method="POST">
    <input type="hidden" name="_method" value="DELETE">
</form>
```

The form in Example 3-34, since it's passing Laravel the method of "DELETE," will match routes defined with Route::delete but not those with Route::post.

CSRF protection

If you've tried to create and submit a form in a Laravel application already, you've likely run into the dreaded TokenMismatchException. If you run the form in Example 3-34, you'll actually run into this exception already.

By default, every route in Laravel except "read-only" routes (those using GET, HEAD, or OPTIONS) are protected against Cross-Site Request Forgery by requiring a token (in the form of an input named _token) be passed along with each request. This token is generated at the start of every session, and every non-read-only route compares the submitted _token against the session token.

You have two options for getting around this. The first, and preferred method, is to add the _token input to each of your submissions. In HTML forms, that's simple; look at Example 3-35.

Example 3-35. Form method spoofing

```
<form action="/tasks/5" method="POST">
    <input type="hidden" name="_method" value="DELETE">
    <input type="hidden" name="_token" value="{{ csrf_token() }}">
</form>
```

In JavaScript applications, it's a bit more work, but not much. The most common solution, for sites using jQuery, is to store the token in a meta tag on every page like in Example 3-36.

Example 3-36. Storing the CSRF token in a meta tag

```
<meta name="csrf-token" content="{{ csrf_token() }}">
```

Storing the token in a meta tag makes it easy to globally bind that to the correct HTTP header, which you can do once globally for all jQuery requests, like in Example 3-37.

Example 3-37. Globally binding a jQuery header for CSRF

```
$.ajaxSetup({
    headers: {
        'X-CSRF-TOKEN': $('meta[name="csrf-token"]').attr('content')
    }
});
```

Laravel will check the `X-CSRF-TOKEN` on every request and valid tokens passed there will mark the CSRF protection as satisfied.

Redirects

There are three common responses that you'll return from your controller methods or route Closures: views, redirects, and errors. We've already covered views and we'll cover errors next, but let's address redirects.

There are two common ways to generate a redirect; we'll use the Façades here, but you may prefer the global helper. Both are create an instance of `Illuminate\Http\RedirectResponse`, performing some convenience methods on it, and then returning it; you could do this manually, but you'll have to do a little more work yourself.

Example 3-38. Three ways to return a redirect

```
Route::get('redirect-with-facade', function () {
    return Redirect::to('auth/login');
});

Route::get('redirect-with-helper', function () {
    return redirect()->to('auth/login');
});

Route::get('redirect-with-helper-shortcut', function () {
    return redirect('auth/login');
});
```

Note that the `redirect()` helper exposes the same methods as the `Redirect` Façade, but it also has a shortcut; if you pass parameters directly to the helper, instead of chaining methods after it, it's a shortcut to the `to()` Redirect method.

You'll notice `to()`, the most commonly used redirect method, has a first parameter that should be set to the URI that you want to redirect the user to. There are a few other options available, though.

Redirect to

The method signature for the `to()` method for redirects looks like this:

```
function to($to = null, $status = 302, $headers = [], $secure = null)
```

`$to` is a valid internal URI; `$status` is the HTTP status (defaulting to `301 FOUND`); `$headers` allows you to define which HTTP headers to send along with your redirect; and `$secure` allows you to override the default choice of `http` vs `https` (which is normally set based on your current request URL).

Redirect route

The `route()` method is the same as the `to()` method, but rather than point to a particular URI, it points to a particular route name.

Example 3-39. Redirect route

```
Route::get('redirect', function () {
    return Redirect::route('conferences.index');
});
```

Note that, since some route names require parameters, its parameter order is a little bit different; it has an optional second parameter for the route parameters:

```
function route($to = null, $parameters = [], $status = 302, $headers = [])
```

So, using it might look a little like Example 3-40.

Example 3-40. Redirect route with parameters

```
Route::get('redirect', function () {
    return Redirect::route('conferences.show', ['conference' => 99]);
});
```

Redirect back

Because of some of the built-in conveniences of Laravel's session implementation, your application will always have a knowledge of what the user's previously-visited page was. That opens up the opportunity for a `Redirect::back()` redirect, which simply redirects the user to whatever page they came from.

Redirect guest and intended

When a user visits a page they're currently not authenticated for—for example, visiting a dashboard when their login session has expired—Laravel captures their **intended** URI and redirects back to it after a successful authentication. This is performed using `guest()` and `intended()`.

`Redirect::guest()` is a normal `Redirect::to()` redirect, except it captures the current URL in a query parameter named "url.intended" for use later. You would use this to redirect a user *away* from their current URL with the intent for them to return after authentication.

`Redirect::intendend()` grabs the `url.intended` query parameter and redirects to it. You would use this after successfully authenticating a user, to redirect them back to their intended URI.

Thankfully, the baked-in Laravel authentication already handles these both for you, but you can use them manually if you're doing your own authentication.

Other redirect methods

The redirect service provides other methods that are less commonly used, but still available:

- `home()` redirects to a route named `home`
- `refresh()` redirects to the same page the user is currently on
- `away()` allows for redirecting to an external URL without the default URL validation
- `secure()` is like `to()` with the `secure` parameter set to `true`
- `action()` allows you to like to a controller and method like this: `action(`*`MyController@myMethod`*`)`

Redirect with

When you're redirecting the user to a different page, you often want to pass certain data along with them. You could manually flash the data to the session, but Laravel has some convenience methods to help you with that.

Most commonly, you can pass along either an array of keys and values or a single key and value using `with()`, like in example Example 3-41.

Example 3-41. Redirect with data

```
Route::get('redirect-with-key-value', function () {
    return Redirect::to('dashboard')
        ->with('error', true);
});

Route::get('redirect-with-array', function () {
    return Redirect::to('dashboard')
        ->with(['error' => true, 'message' => 'Whoops!']);
});
```

You can also redirect with the form input flashed; this is most common in the case of a validation error, where you want to send the user back to the form they just came from.

Example 3-42. Redirect with form input

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

Route::post('form', function () {
    return Redirect::to('form')
        ->withInput()
        ->with(['error' => true, 'message' => 'Whoops!']);
});
```

The easiest way to get the flashed Input that was passed with `withInput` is the `old()` helper, which can be used to get all old input (`old()`) or just the value for a particular key (`old(`*`username`*`)`). You'll commonly see this in views, which allows this HTML to be used both on the "create" and the "edit" view for this form:

```
    <input name="username" value="<?=
        old('username', 'Default username instructions here');
    ?>">
```

Speaking of validation, there is also a useful helper for passing errors along with a redirect response. You can pass it any "provider" of errors, which may be an error string, an array of errors, or, most commonly, the Illuminate Validator, which we'll cover in chapter ???.

Example 3-43. Redirect with errors

```
Route::post('form', function () {
    $validator = Validator::make($request->all), $this->validationRules);

    if ($validator->fails()) {
        return Redirect::to('form')
```

```
            ->withErrors($validator)
            ->withInput();
    }
});
```

`withErrors()` automatically shares an `$errors` variable with the views of the page it's redirecting to, for you to handle however you'd like.

Abort

After returning views and redirects, the most common way to exit a route is to abort. There's a globally available `abort()` method which optionally takes an HTTP status code, a message, and a headers array as parameters.

Example 3-44. 403 Forbidden Abort

```
Route::post('something-you-cant-do', function () {
    abort(403, 'You cannot do that!');
});
```

Custom responses

There are a few other options available for us to return, so let's go over the most common responses after views, redirects, and errors. Just like with redirects, you can either use the `Response` Façade or the `response()` helper to run these methods on.

Response make

If you want to create an HTTP Response manually, just pass your data into the first parameter of `Response::make()`: `return Response::make(`*`Hello, World!`*`)`. Once again, the second parameter is the HTTP status code and the third is your headers.

Response json/jsonp

To create a JSON-encoded HTTP response manually, pass your JSON-able content (arrays, collections, or whatever else) to the `json()` method: `return Response::json(User::all());`. It's just like `make()`, except it `json_encode`s your content and sets the appropriate headers.

Response download

To send a downloadable file, pass either a string filename or a `SplFileInfo` instance to `download()`, with an optional second parameter of the filename: `return Response::download(`*`file501751.pdf, myFile.pdf`*`);`.

Testing

In some other communities, the idea of testing controller methods is common, but within Laravel (and most of the PHP community), it's most common to rely on *application testing* to test the functionality of routes.

For example, to test to make sure a `POST` route works correctly, you can write a test like Example 3-45.

Example 3-45. Writing a simple `POST` route test

```php
// AssignmentTest.php
public function test_post_creates_new_assignment()
{
    $this->post('/assignments', [
        'title' => 'My great assignment'
    ]);

    $this->seeInDatabase('assignments', [
        'title' => 'My great assignment'
    ]);
}
```

Did we directly call the controller methods? No. But we ensured that the goal of this route—to receive a `POST` and save its important information to the database—was met.

You can also use similar syntax to visit a route and verify that certain text shows up on the page, or that clicking certain buttons do certain things.

Example 3-46. Writing a simple `GET` route test

```php
// AssignmentTest.php
public function test_list_page_shows_all_assignments()
{
    $assignment = Assignment::create([
        'title' => 'My great assignment'
    ]);

    $this->visit('assignments')
        ->andSee(['My great assignment']);
}
```

TL;DR

Laravel's routes are defined in `app/Http/routes.php`, where you can define the expected URI, which segments are static and which are parameters, which HTTP verbs can access it, and how to resolve it. You can also attach middleware to routes, group them, and give them names.

What is returned from the route Closure or controller method dictates how Laravel response to the user. If it's a string or a view, it's presented to the user; if it's other sorts of data, it's converted to JSON and presented to the user; and if it's a redirect, it forces a redirect.

Laravel provides a series of tools and conveniences to simplify common routing-related tasks and structures. These include resource controllers, route model binding, and form method spoofing.

@todo: Need to add Laravel 5.2's middleware groups to this chapter

CHAPTER 4

Blade Templating

Compared to most other backend languages, PHP actually functions relatively well as a templating language. But it has its shortcomings, and it's also just ugly to be be using `<?php` inline all over the place, so you can expect most modern frameworks to offer a templating language.

Unlike many other Symfony-based frameworks, Laravel doesn't use Twig by default—although there's a Twig Bridge package that makes it easy to use Twig if you like it.

Instead, Laravel provides a custom templating engine called Blade, which is inspired by .NET's Razor engine. It's functionally very similar to Twig, but the syntax is closer to Razor and the learning curve for PHP developers tends to be lower than for Twig.

Take a look at a common display pattern in PHP, Twig, and Blade in Example 4-1.

Example 4-1. PHP vs. Twig vs. Blade

```
<?php /* PHP */ ?>
<?php if (empty($users)): ?>
    No users.
<?php else: ?>
    <?php foreach ($users as $user): ?>
        • <?= $user->first_name ?> <?= $user->last_name ?><br>
    <?php endforeach; ?>
<?php endif; ?>

{# Twig #}
{% for user in users %}
    • {{ user.first_name }} {{ user.last_name }}<br>
{% else %}
    No users.
{% endfor %}
```

```
{{-- Blade --}}
@forelse ($users as $user)
    • {{ $user->first_name }} {{ $user->last_name }}<br>
@empty
    No users.
@endforelse
```

As you can see, Blade's syntax tends to be somewhere between PHP and Twig—it's more powerful, like Twig, and has convenience helpers like `forelse`, but its syntax is closer to PHP than Twig.

Additionally, since all Blade syntax is compiled into normal PHP code and then cached, it's fast and it allows you to use native PHP in your Blade files if you want. The common recommendation, however, is to keep any PHP tags out of your Blade files.

Echoing data

As you can see in the examples above, `{{` and `}}` are used to wrap sections of PHP that you'd like to echo. `{{ $variable }}` is similar to `<?= $variable ?>` in plain PHP.

It's different in one way, however: Blade escapes all echoes by default using PHP's `htmlentities`. That means `{{ $variable }}` is functionally equivalent to `<?= htmlentities($variable) ?>`. If you want to echo without the escaping, use `{!!` and `!!}` instead.

{{ and }} when using a frontend templating framework

You might've noticed that the *echo* syntax for Blade (`{{`/`}}`) is similar to the *echo* syntax for many frontend frameworks. So, how does Blade know when you're writing Blade vs. Handlebars?

Any `{{` that's prefaced with an `@` will be ignored by Blade. So in Example 4-2, the first would be parsed by Blade and the other would be echoed out directly.

Example 4-2. Using `@{{` to ask Blade to skip

```
// Parsed as Blade
{{ $bladeVariable }}

// @ removed, and echoed to the view directly
@{{ handlebarsVariable }}
```

Control structures

Most of the control structures in Blade will be very familiar. Many directly echo the name and structure of the same tag in PHP.

There are a few convenience helpers, but in general, the control structures primarily just look cleaner than they would in PHP.

Conditionals

@if

Blade's `@if ($condition)` compiles to `<?php if ($condition): ?>`. `@else`, `@elseif`, and `@endif` also compile to the exact same syntax in PHP. Take a look at Example 4-3 for an example.

Example 4-3. @if, @else, @elseif, and @endif

```
@if (count($talks) === 1)
    There is one talk at this time period.
@elseif (count($talks) === 0)
    There are no talks at this time period.
@else
    There are {{ count($talks) }} talks at this time period.
@endif
```

Just like with the native PHP conditionals, you can mix and match these how you want. They don't have any special logic; there's literally a parser looking for something with the shape of `@if ($condition)` and replacing it with the appropriate PHP code.

@unless and @endunless

`@unless`, on the other hand, is a new syntax that doesn't have a direct cognate in PHP. It's the exactly the same as `@if`, but the inverse. `@unless ($condition)` is the same as `<?php if (! $condition)`. See it in use in Example 4-4.

Example 4-4. @unless and @endunless

```
@unless ($user->hasPaid())
    You can complete your payment by switching to the payment tab.
@endunless
```

Loops

@for, @foreach, and @while

`@for`, `@foreach`, and `@while` work the same in Blade as they do in PHP.

Example 4-5. @for and @endfor

```
@for ($i = 0; $i < $talk->slotsCount(); $i++)
    The number is {{ $i }}
@endfor
```

Example 4-6. @foreach and @endforeach

```
@foreach ($talks as $talk)
    • {{ $talk->title }} ({{ $talk->length }} minutes)
@endforeach
```

Example 4-7. @while

```
@while ($item = array_pop($items))
    {{ $item->orSomething() }}<br>
@endwhile
```

@forelse

We've already looked at it in the introduction, but `@forelse` is a `@foreach` that also allows you to program in a fallback if the object you're iterating over is empty.

Example 4-8. @forelse

```
@forelse ($talks as $talk)
    • {{ $talk->title }} ({{ $talk->length }} minutes)
@empty
    No talks this day.
@endforelse
```

Or

If you're ever unsure whether a variable is set, you're probably used to checking `isset()` on it before echoing it, and echoing something else if it's not set. Blade has a convenience helper, `or`, that does it for you and lets you set a default fallback: `{{ $title or "Default" }}` will echo the value of `$title` if it's set, or "Default" if not.

Template inheritance

Just like Twig, Blade provides a structure for inheritance that allows views to extend, modify, and include other views.

Here's how inheritance is structured with Blade.

Defining sections with @section/@show and @yield

Let's start with a top-level Blade layout, like in Example 4-9. This is the definition of the generic page wrapper that we'll later place page-specific content into.

Example 4-9. Blade layout

```
<!-- resources/views/layouts/master.blade.php -->
<html>
    <head>
        <title>My Site | @yield('title', 'Home Page')</title>
    </head>
    <body>
        <div class="container">
            @yield('content')
        </div>
        @section('footerScripts')
            <script src="app.js">
        @show
    </body>
</html>
```

This looks a bit like a normal HTML page, but you can see we've `yielded` in two places (*title* and *content*), and we've defined a *section* in a third (*footerScripts*).

We have three Blade directives here that each look a little different: `yield(`*`content`*`)` alone, `yield(`*`title, \'Home Page\`*`)` with a defined default, and `@section … @show` with actual content in it.

All three function essentially the same. All three are defining that there's a section with a given name (which is the first parameter). All three are defining that the section **can** be extended later. And all three are telling what to do if the section isn't extended, either by providing a string fallback (*Home Page*), no fallback (which will just not show anything if it's not extended), or an entire block fallback (in this case, `<script src="app.js">`).

What's different? Well, clearly, `yield(`*`content`*`)` has no default content. But additionally, the default content in `yield(`*`title`*`)` *only* will be shown if it's never extended. If it is extended, its child sections will not have programmatic access to the default value. `@section … @show`, on the otherhand, is both defining a default *and* doing so in a way that its default contents will be available to its children, through `@parent`.

Once you have a parent layout like this, you can extend it like in Example 4-10.

Example 4-10. Extending a Blade Layout

```
<!-- resources/views/dashboard.blade.php -->
@extends('layouts.master')
```

```
@section('title', 'Dashboard')

@section('content')
    Welcome to your application dashboard!
@endsection

@section('footerScripts')
    @parent

    <script src="dashboard.js">
@endsection
```

This child view will actually allow us to cover a few new concepts in Blade inheritance.

@extends

First, with `@extends('layouts.master')`, we define that this view should not be rendered on its own, but that it instead *extends* another view. That means its role is to define the content of various sections, but not to stand alone. It's almost more like a series of buckets of content, rather than an HTML page. It also defines that the view it's extending lives at `resources/views/layouts/master.blade.php`.

@section and @endsection

Second, with `@section('title', 'Dashboard')`, we provide our content for the first section, *title*. Since the content is so short, instead of using `@section` and `@endsection`, we're just using a shortcut. This allows us to pass the content in as the second parameter of `@section` and then move on. If it's a bit disconcerting to see `@section` without `@endsection`, you could just use the normal syntax.

Third, with `@section('content')` and on, we use the normal syntax to define the contents of the *content* section. We'll just throw a little greeting in for now. Note, however, that when you're using `@section` in a child view, you end it with `@endsection` (or its alias `@stop`), instead of `@show`, which is reserved for defining sections in parent views.

@parent

Fourth, with `@section('footerScripts')` and on, we use the normal syntax to define the contents of the *footerScripts* section.

But remember, we actually defined that content (or, at least, its "default") already in the master layout. So this time, we have two options: we can either *overwrite* the content from the parent view, or we can *add* to it.

You can see that we have the option to include the content from the parent by using the `@parent` directive within the section. If we didn't, the content of this section would entirely overwrite anything defined in the parent for this section.

@include

So, we have the basics of inheritance established. There are a few more tricks we can perform.

What if we're in a view and want to pull in another view? Maybe we have a call-to-action "Sign up" button that we want to re-use around the site. And maybe we want to customize its button text every time we use it. Take a look at Example 4-11.

Example 4-11. Including view partials with @include

```
<!-- resources/views/home.blade.php -->
<div class="content" data-page-name="{{ $pageName }}">
    <p>Here's why you should sign up for our service: <strong>It's Great.</strong></p>

    @include('sign-up-button', ['text' => 'See just how great it is'])
</div>

<!-- resources/views/sign-up-button.blade.php -->
<a class="button button--callout" data-page-name="{{ $pageName }}">
    <i class="exclamation-icon"></i> {{ $text }}
</a>
```

`@include` pulls in the partial and, optionally, passes data into it. Note that, not only can you *explicitly* pass data via the second parameter of `@include`, you can also reference any variables that are available to the including view (`$pageName`, in this example). Once again, you can do whatever you want, but I would recommend you consider always passing every variable explicitly that you intend to use, just for clarity.

@each

You can probably imagine some circumstances in which you'd need to loop over an array or collection and `@include` a partial for each item. There's a directive for that.

Let's say we have a sidebar composed of modules, and we want to incude multiple modules, each with a different title. Take a look at Example 4-12.

Example 4-12. Using view partials in a loop with @each

```
<!-- resources/views/sidebar.blade.php -->
<div class="sidebar">
    @each('partials.module', $modules, 'module', 'partials.empty-module')
```

```
</div>

<!-- resources/views/partials/module.blade.php -->
<div class="sidebar-module">
    <h1>{{ $module->title }}</h1>
</div>

<!-- resources/views/partials/module.blade.php -->
<div class="sidebar-module">
    No modules :(
</div>
```

Take a look at that `@each` syntax. The first parameter is the name of the view partial. The second is the array or collection to iterate over. The third is the variable name that each item will be passed to the view as. And the optional fourth parameter is the view to show if the array or collection is empty.

View composers and service injection

Like we covered in Chapter 3, we can pass data to our views from the route definition (see Example 4-13).

Example 4-13. Reminder on how to pass data to views

```
Route::get('passing-data-to-views', function () {
    return view('dashboard')
        ->with('key', 'value');
});
```

There are times, however, when you will find yourself passing the same data over and over to multiple views. Or, you might find yourself using a header partial or something else similar that requires some data; will you now have to pass that data in from every route definition that might ever load that header partial?

Binding data to views using view composers

Thankfully, there's a simpler way. The solution is called a view composer, and it allows you to define that *any time a particular view loads, it should have certain data passed to it*--without the route definition having to pass that data in explicitly.

Let's say you have a sidebar on every page which is defined in a partial named `partials.sidebar` and then included on every page. This sidebar shows a list of the last seven posts that were published on your site. If it's on every page, every route definition would normally have to grab that list and pass it in, like in Example 4-14.

Example 4-14. Passing sidebar data in from every route

```
Route::get('home', function () {
    return view('home')
        ->with('posts', Post::recent());
});

Route::get('about', function () {
    return view('about')
        ->with('posts', Post::recent());
});
```

That could get annoying quickly. Instead, we're going to use view composers to "share" that variable with a prescribed set of views. We can do this a few ways, so let's start simple and move up.

Sharing a variable globally

First, the simplest: Just globally "share" a variable like in Example 4-15:

Example 4-15. Sharing a variable globally

```
view()->share('posts', Post::recent());
```

You'll likely place this code in some form of custom `ViewComposerServiceProvider` (see ??? to learn more about Service Providers), but for now you could also just put it in `App\Providers\AppServiceProvider` in the `boot` method.

Using `view()->share()` makes the variable accessible to every view in the entire application, however, so it might be overkill.

Closure-based view composers

The next option is to use a Closure-based view composer to share variables with a single view like in Example 4-16.

Example 4-16. Creating a Closure-based view composer

```
view()->composer('partials.sidebar', function ($view) {
    $view->with('posts', Post::recent());
});
```

As you can see, we've defined the name of the view we want it shared with (`partials.sidebar`) in the first parameter and then passed a Closure to the second parameter; in the Closure, we've used `$view->with()` to share a variable, but now only with a specific view.

View composers for multiple views

Anywhere a view composer is binding to a particular view (like in Example 4-16 which binds to `partials.sidebar`), you can also pass an array of view names instead to bind to multiple views.

Or, you can use an asterisk in the view path: `partials.*`, or `tasks.*`, or just `*`.

Class-based view composers

Finally, the most flexible but also most complex option is to create a dedicated class for your view composer.

First, let's create the view composer class. There's no formally defined place for view composers to live, but the docs recommend `App\Http\ViewComposers`. So let's create `App\Http\ViewComposers\RecentPostsComposer` like in Example 4-17.

Example 4-17. A view composer

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

use App\Post;
use Illuminate\Contracts\View\View;

class RecentPostsComposer
{
    private $posts;

    public function __construct(Post $posts)
    {
        $this->posts = $posts;
    }

    public function compose(View $view)
    {
        $view->with('posts', $this->posts->recent());
    }
}
```

As you can see, we're injecting the `Post` model (type-hinted constructor parameters of view composers will be automatically injected; see ??? for more on the container and dependency injection). Note that we could skip the `private $posts` and the constructor injection and just used `Post::recent()` in the `compose` method if we wanted. Then when this composer is called, it runs the `compose` method, in which we bind the *posts* variable to the result of the `recent()` method.

Just like the other methods of sharing variables, this view composer needs to have a binding somewhere. Again, you'd likely create a custom `ViewComposerServicePro` vider, but for now we'll just put it in the boot method of App\Providers\AppServiceProvider.

Example 4-18. Registering a view composer in AppServiceProvider

```
view()->composer(
    'partials.sidebar',
    'App\Http\ViewComposers\RecentPostsComposer'
);
```

Note that this binding is the same as a Closure-based view composer, but instead of passing a Closure, we're passing the class name of our view composer. Now, every time Blade renders the `partials.sidebar` view, it'll automatically run our provider and pass the view a `posts` variable set to the results of the `recent()` method on our Post model.

Service injection

There are three primary types of data we're most likely to inject into a view: collections of data to iterate over, single objects that you're displaying on the page, and services that generate data or views.

With a service, the pattern will most likely look like Example 4-19, where we inject an instance of the service into the route definition by type-hinting it in the route definition's method signature, and then pass it into the view.

Example 4-19. Injecting services into a view via the route definition constructor

```
Route::get('injecting', function (AnalyticsService $analytics) {
    return view('injecting')
        ->with('analytics', $analytics);
});
```

Just as view composers, Blade's service injection offers a convenient shortcut to reducing duplication in your route definitions. Normally the content of a view using the navigation service above might look like Example 4-20.

Example 4-20. Using an injected navigation service in a view

```
<div class="finances-display">
    {{ $analytics->getBalance() }} / {{ $analytics->getBudget() }}
</div>
```

Blade service injection makes it easy to inject an instance of a class out of the container directly from the view, like in Example 4-21.

Example 4-21. Injecting a service directly into a view

```
@inject('analytics', 'App\Services\Analytics')

<div class="finances-display">
    {{ $analytics->getBalance() }} / {{ $analytics->getBudget() }}
</div>
```

As you can see, this `@inject` method has actually made an `$analytics` variable available, which we're using later in our view.

The first parameter of `@inject` is the name of the variable you're injecting, and the second parameter is the class or interface that you want to inject an instance of. This is resolved just like when you type-hint a dependency in a constructor elsewhere in Laravel, and if you're unfamiliar with how that works, take a look at ??? to learn more.

Just like view composers, Blade service injection makes it easy to make certain data or functionality available to every instance of a view, without having to inject it via the route definition every time.

Custom Blade directives

All of the built-in syntax of Blade that we've covered so far—`@if`, `@unless`, etc.--are called Blade *directives*. Each Blade directive is a mapping between a pattern (e.g. `@if ($condition)`) and a PHP output (e.g. `<?php if ($condition): ?>`).

Directives aren't just for the core; you can actually create your own. You might think directives are good for making little shortcuts to bigger pieces of code—for example, `@button('buttonName')`, and having it expand to a larger set of button HTML. This isn't a *terrible* idea, but for simple code expansion like this you might be better off including a view partial.

I've found custom directives the most useful when they simplify some form of repeated logic. Let's say we were tired of having to wrap our code with `@if (Auth::guest())` (to check if a user is logged in or not) and we wanted a custom `@ifGuest` directive.

As with view composers, it might be worth having a custom Service Provider to register these, but for now let's just put it in the `boot` method of `App\Providers\AppServiceProvider`. Take a look at Example 4-22 to see what this binding will look like.

Example 4-22. Binding a custom Blade directive

```
// AppServiceProvider
public function boot()
{
    Blade::directive('isGuest', function () {
        return "<?php if (Auth::guest()): ?>";
    });
}
```

We've now registered a custom directive `@isGuest`, which will be replaced with the PHP code `<?php if (Auth::guest()): ?>`.

This might feel strange. You're writing a *string* that will be returned and then executed as PHP. It takes a minute to get your brain wrapped around it, but once you do you can see how powerful it can be.

> You might be tempted to do some logic to make your custom directive faster by performing an operation *in* the binding and then embedding the result within the returned string:
>
> ```
> Blade::directive('isGuest', function () {
>     // Anti-pattern! Do not copy.
>     $isGuest = Auth::guest();
>     return "<?php if ({$isGuest}): ?>";
> });
> ```
>
> The problem with this idea is that it assumes this directive will be re-created on every page load. However, Blade caches aggressively, so you're going to find yourself in a bad spot if you try this.

Parameters in custom Blade directives

What if you want to check a condition in your custom logic? Check out Example 4-24.

Example 4-23. Creating a Blade directive with parameters

```
// Binding
Blade::directive('newlinesToBr', function ($expression) {
    return "<?php echo nl2br{$expression}; ?>";
});

// In use
<p>@newlinesToBr($message->body)</p>
```

The `$expression` parameter received by the Closure represents whatever's within the parentheses *and the parentheses themselves*. So, in Example 4-23, `$expression` is actually `($message->body)`. That's why there are no parentheses after `nl2br` in the binding; they're already included with `$expression`.

So, if you find yourself constantly writing the same conditional logic over and over, consider a Blade directive.

Using custom Blade directives for a multitenant app

Let's imagine we're building an application that supports *multitenancy*, which means you might be visiting the site from `www.myapp.com`, `client1.myapp.com`, `client2.myapp.com`, or whatever else.

Let's imagine we have written a class to encapsulate some of our multietnancy logic and named it `Context`. This class will captures information and logic about the context of the current visit: who's the authenticated user? Which subdomain are we visiting? And, important to this example: are we "public" (`www.myapp.com`) or "client" (`someClientName.myapp.com`)?

We'll probably frequently resolve that `Context` class in our views and performing conditionals on it, like Example 4-24. The `app(`*`context`*`)` is a shortcut to get an instance of a class from the container, which we'll learn more about in ???.

Example 4-24. Conditionals on Context without a custom Blade directive

```
@if (app('context')->isPublic())
    &copy; Copyright MyApp LLC
@else
    &copy; Copyright {{ app('context')->client->name }}
@endif
```

What if we could simplify the `@if (app('context')->isPublic())` to just `@ifPublic`? Let's do it. Check out Example 4-25.

Example 4-25. Conditionals on Context with a custom Blade directive

```
// Binding
Blade::directive('ifPublic', function () {
    return "<?php if (app('context')->isPublic()): ?>";
});

// In use
@ifPublic
    &copy; Copyright MyApp LLC
@else
    &copy; Copyright {{ app('context')->client->name }}
@endif
```

Since this resolves out to a simple `if` statement, we can still rely on the native `@else` and `@endif` conditionals. But if we wanted, we could also create a custom `@elseIfClient` directive, or a separate `@ifClient` directive, or really whatever else we want.

Testing

Testing views is not common, but it's possible. The most common method of testing views is through application testing, meaning that you're actually calling the route that displays the views, and ensuring the views have certain content. You can also click buttons or submit forms and ensure that you are redirected to a certain page, or that you see a certain error. Learn more in ???.

Example 4-26. Testing that a view displays certain content

```
// EventsTest.php
public function test_list_page_shows_all_events()
{
    $event1 = factory(Event::class)->create();
    $event2 = factory(Event::class)->create();

    $this->visit('events')
        ->andSee($event1->title)
        ->andSee($event2->title);
}
```

TL;DR

Blade is Laravel's templating engine. It's a little bit like Twig and a little bit like straight PHP. Its "safe echo" brackets are `{{` and `}}`, its unprotected echo brackets are `{!!` and `!!}`, and it has a series of directives that all begin with `@` (`@if` and `@unless`, for example).

Define a parent template and leave "holes" in it for content using `@yield` and `@section/@show`. Teach its child views to extend it using `@extends('parent.view.name')`, and define their sections using `@section/@endsection`. Use `@parent` to reference the content of the same block in the parent.

View composers make it easy to define that, every time a particular view or subview loads, it has certain information available to it. And service injection allows the view itself to dictate what data it needs.

CHAPTER 5
View Components

Laravel is primarily a PHP framework, but it also has a series of components primarily focused on generating frontend code. Some of these, like pagination and message bags, are PHP helpers that target the frontend, but Laravel also provides a Gulp-based build system called Elixir and some conventions around non-PHP assets.

Since Elixir is at the core of the non-PHP frontend components, let's start there.

Elixir

Elixir (not to be confused with the functional programming language) is a build tool that provides a simpler user interface and a series of conventions on top of Gulp. While it's far less complex than a framework, it performs the same task of simplifying and organizing best practices around a tool—in Laravel's case, the tool is PHP, and in Elixir's case, the tool is Gulp.

A quick introduction to Gulp

Gulp is a JavaScript tool designed for running asset compilation and other steps of your build process.

Gulp is similar to Grunt, Rake, or make—it allows you to define an action or series of actions to take every time you build your application. This will commonly include running a CSS preprocessor like Sass or Less, copying files, concatenating and minifying JavaScript, and much more.

Gulp, and therefore Elixir, are based on the idea of streams. The beginning of most tasks will load some files into the stream buffer, and then the task will apply transformations to the content—preprocess it, minify it, and then maybe save the content to a new fine.

At its core, Elixir is just a tool in your Gulp toolbox. There isn't even such a thing as an Elixir file; you'll define your Elixir tasks in your `gulpfile.js`. But it looks a lot different than vanilla GUlp tasks, and you'll have to do a lot less work to get it running out of the box.

Elixir's core feature is simplifying the most common Gulp tasks by means of a simpler API and a series of naming and application structure conventions.

Let's look at a common example: Running Sass to pre-process your CSS styles. In a normal Gulp environment, that might look a little bit like Example 5-1.

Example 5-1. Compiling a Sass file in Gulp

```
var gulp = require('gulp'),
    sass = require('gulp-ruby-sass'),
    autoprefixer = require('gulp-autoprefixer'),
    rename = require('gulp-rename'),
    notify = require('gulp-notify'),
    livereload = require('gulp-livereload'),
    lr = require('tiny-lr'),
    server = lr();

gulp.task('sass', function() {
    return gulp.src('resources/assets/sass/app.scss')
        .pipe(sass({
            style: 'compressed',
            sourcemap: true
        }))
        .pipe(autoprefixer('last 2 version', 'ie 9', 'ios 6'))
        .pipe(gulp.dest('public/css'))
        .pipe(rename({suffix: '.min'}))
        .pipe(livereload(server))
        .pipe(notify({
            title: "Karani",
            message: "Styles task complete."
        }));
});
```

Now, I've seen worse. It reads well and you know what's going on. But there's a *lot* happening that you'll just pull into every site you ever make. It can get confusing and repetitive.

Let's try that same task in Elixir in Example 5-2.

Example 5-2. Compiling a Sass file in Gulp

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

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

That's it. That covers all the basics—preprocessing, notification, folder structure, auto-prefixing and much more.

Elixir folder structure

Much of Elixir's simplicity comes from the assumed directory structure. Why make the decision fresh in every new application about where the source and compiled assets live? Just stick with Elixir's convention and you won't have to think about it ever again.

Every new Laravel app comes with a `resources` folder with an `assets` subfolder, which is where Elixir will expect your frontend assets to live. Your Sass will live in `resources/assets/sass`, or your Less would live in `resoures/assets/less`, and your JavaScript would live in `resources/assets/js`. These would export to `public/css` and `public/js`.

But if you're interested in changing the structure, you can always change the source and public paths by changing the appropriate properties (`assetsPath` and `publicPath`) on the `elixir.config` object.

Running Elixir

Since Elixir runs on Gulp, you'll need to set up a few tools first.

1. First, you'll need Node.js installed. Visit the Node web site to learn how to get it running.
2. Next, you'll need to install Gulp globally on your machine. Just run `npm install --global gulp` from the terminal anywhere on your machine.

   Once Node and Gulp are installed, you will never have to run those commands again. Now you're ready to install this project's dependencies.
3. Now you can open the project root in your terminal, and run `npm install` to install the required packages (Laravel ships with an Elixir-ready `package.json` file to direct npm).

You're now set up! You can run `gulp` to run Gulp/Elixir once, `gulp watch` to run it every time you make any file changes, or `gulp scripts` or `gulp styles` to just run the script or style tasks.

What does Elixir provide?

We've already covered that Elixir can preprocess your CSS using Sass or Less. It can concatenate files, minify them, rename them, and copy them, and it can copy entire directories or individual files.

Elixir can also process CoffeeScript, ES6 JavaScript, and run Browserify and Autoprefixer on your code. Not only can it, but most of the modern coding standards for JavaScript and CSS are covered on every script or style, out of the box.

Elixir can also run your tests. There's a method for PHPUnit and one for PHPSpec; both listen to changes to your test files and re-run your test suite every time you make any changes.

The Elixir documentation covers all of these options and more, but we'll cover a few specific use cases in the following sections.

The Production Flag

By default, Elixir doesn't minify all the files it's generating. But if you want to run the build scripts in "production" mode, with all minification enabled, add the `--production` flag:

```
$ gulp --production
```

Passing multiple files

Most of the Elixir methods that normally accept a single file (e.g. `mix.sass('app.scss')`) can also take an array of files like in Example 5-3.

Example 5-3. Compiling multiple files with Elixir

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

elixir(function(mix) {
    mix.sass([
        'app.scss',
        'public.scss'
    ]);
});
```

Source maps

By default, Elixir generates source maps for your files—you'll see them as a `.{file name}.map` file next to each generated file. If you're not familiar with source maps, they work with any sort of preprocessor to teach your browser's web inspector which files generated the compiled source you're inspecting.

If you don't want source maps, you can always change the configuration before your elixir block like in Example 5-4.

Example 5-4. Disabling source maps in Elixir

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

elixir.config.sourcemaps = false;

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

Preprocessor-less CSS

If you don't want to deal with a preprocessor, there's a command for that—it will grab all of your CSS files, concatenate them, and output them to the `public/css` directory, just as if they had been run through a preprocessor. If you don't specific an ouput file name, it'll end up in `all.css`. There are a few options, which you can see in Example 5-5.

Example 5-5. Combining Stylesheets with Elixir

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

elixir(function(mix) {
    // Combines all files from resources/assets/css and subfolders
    mix.styles();

    // Combines files from resources/assets/css
    mix.styles([
        'normalize.css',
        'app.css'
    ]);

    // Combines all styles from other directory
    mix.stylesIn('resources/some/other/css/directory');

    // Combines given styles from resources/assets/css
    // and outputs to a custom directory
    mix.styles([
        'normalize.css',
        'app.css'
    ], 'public/other/css/output.css');

    // Combines given styles from custom directory
    // and outputs to a custom directory
    mix.styles([
        'normalize.css',
        'app.css'
    ], 'public/other/css/output.css', 'resources/some/other/css/directory');
});
```

Concatenating JavaScript

The options available for working with normal JavaScript files are very similar to those available for normal CSS files. Take a look at Example 5-6. Like with `styles()`, any commands not provided with an output filename will generate to `public/js/all.js`.

Example 5-6. Combining JavaScript files with Elixir

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

elixir(function(mix) {
    // Combines files from resources/assets/js
    mix.scripts([
        'jquery.js',
        'app.js'
    ]);

    // Combines all scripts from other directory
    mix.scriptsIn('resources/some/other/js/directory');

    // Combines given scripts from resources/assets/js
    // and outputs to a custom directory
    mix.scripts([
        'jquery.js',
        'app.js'
    ], 'public/other/js/output.js');

    // Combines given scripts from custom directory
    // and outputs to a custom directory
    mix.scripts([
        'jquery.js',
        'app.js'
    ], 'public/other/js/output.js', 'resources/some/other/js/directory');
});
```

Versioning

Most of the tips from Steve Souder's Even Faster Web Sites have made their way into our everyday development practices. We move scripts to the footer, reduce the number of HTTP requests, and more, often without even realizing where those ideas originated.

But one of Steve's tips is still very uncommon, and that is setting a very long cache life on assets (scripts, styles, and images). This means there will be less requests to your server to get the latest version of your assets. But it also means that users are extremely likely to have a cached version of your assets, which will make things get outdated, and therefore break, quickly.

The solution to this is versioning. Append a unique hash to each asset's filename *every time you run your build script*, and then that unique file will be cached forever, until the next build.

What's the problem? Well, first you need to get the unique hash generated and appended to your filenames. But you also will need to update your views on every build to reference the new filename.

As you can probably guess, Elixir handles that for you, and it's incredibly simple. There are two components: The versioning tool in Elixir, and the `elixir()` PHP helper. First, you can version your assets by running `mix.version()` like in Example 5-7.

Example 5-7. Mix.version

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

elixir(function(mix) {
    mix.version('public/css/all.css');
});
```

This will now generate a version of that file with a unique hash appended to it—something like `all-84fa1258.css`.

Next, use the PHP `elixir()` helper in your views to refer to that file like in Example 5-8.

Example 5-8. Using the `elixir()` helper in views

```
<link rel="stylesheet" href="{{ elixir("css/all.css") }}">
```

How does Elixir versioning work behind-the-scenes?

Elixir uses `gulp-rev`, which takes care of both appending the hash to the filenames, and also generates a file named `public/build/rev-manifest.json`. This stores the information the `elixir()` helper needs to find the generated file. Take a look at what a sample `rev-manifest.json` looks like:

```
{
  "css/all.css": "css/all-7f592e49.css"
}
```

Tests

With Elixir it's easy to run your PHPUnit or PHPSpec tests every time your test files change.

You have two options, `mix.phpUnit()` and `mix.phpSpec()`, and each will run the frameworks directly from the `vendor` folder, so you won't have to do anything to make them work.

If you add one of these methods to your Gulp file, you'll find they only run once, even if you're using `gulp watch`. How do you get them to respond to changes in your `tests` folder?

There's a separate Gulp command for that: `gulp tdd`. This grabs just the test commands out of your Gulp file, whether `phpUnit()` or `phpSpec()`, listens to the appropriate folder, and re-runs the test suite whenever any files change.

Elixir extensions

Elixir doesn't just provide a simple syntax for its own pre-built tasks, it also makes it easy to define your own.

Let's say you want to save text to a log file at certain points. That's a shell command, which is `echo "message" >> file.log`. Normally we'd define this as a Gulp task, using `shell('echo "message" >> file.log')`, like in Example 5-9.

Example 5-9. Using a Gulp task in Elixir

```
// Define the task
gulp.task("log", function () {
    var message = "Something happened";
    gulp.src("").pipe(shell('echo "' + message + '" >> file.log'));
});

elixir(function (mix) {
    // Use the task in Elixir
    mix.task('log');

    // Bind the task to run every time certain files are changed
    mix.task('log', 'resources/somefiles/to/watch/**/*')
});
```

However, if we want a little more control—for example, if we want to allow you to actually pass in the message, which is really sort of vital to make this particular task work—we can create an Elixir *extension* like in Example 5-10.

Example 5-10. Creating an Elixir extension

```
// Either in gulpfile.js, or in an external file and required in gulpfile.js
var gulp = require("gulp"),
    shell = require("gulp-shell"),
    elixir = require("laravel-elixir");

elixir.extend("log", function (message) {
    new Task('log', function() {
        return gulp.src('').pipe(shell('echo "' + message + '" >> file.log'));
    })
    .watch('./resources/some/files/**/*');
});
```

Pagination

For something that is so common across web applications, pagination still can be wildly complicated to implement. Thankfully, Laravel has a built-in concept of pagination, and it's also hooked into Eloquent results **and** the router by default.

Paginating database results

The most common place you'll see pagination is when you are displaying the results of a database query and there are too many results for a single page. Eloquent and the query builder both read the `page` query parameter from the current query request and use it to provide a `paginate()` method on any result sets; the parameter is how many results you want per page. Take a look at Example 5-11 to see how this works.

Example 5-11. Paginating a Query builder response

```
// PostsController
    public function index()
    {
        return view('posts.index', ['posts' => DB::table('posts')->paginate(20)]);
    }
```

Example 5-11 defines that this route should return 20 posts per page, and will define which page the current user is on based on their URL's `page` query parameter, if it has one. Eloquent models all have the same `paginate()` method.

Manually creating paginators

If you're not working with Eloquent or the Query Builder, **or** if you're working with a complex query (e.g. those using `groupBy`), you might find yourself needing to create a paginator manually. Thankfully, you can do that with the `Illuminate\Pagination\Paginator` or the `Illuminate\Pagination\LengthAwarePaginator` classes.

The difference between the two classes is that `Paginator` will only provide previous and next buttons, but no links to each page; `LengthAwarePaginator` needs to know the length of the full result, so that it can generate links for each individual page.

Both require you to manually extract the subset of content that you want to pass to the view.

Message bags

Another common-but-painful feature in web applications is passing messages between various components of the app, when the end goal is to share them with the user. Your controller, for example, might want to send a validation message: "The `email` field must be a valid e-mail address." However, thatparticular message needs to make it not only to the view layer; it actually needs to survive a redirect and then end up in the view layer of a different page. How do we structure this messaging logic?

`Illuminate\Support\MessageBag` is a class tasked with storing, categorizing, and returning messages that are intended for the end user. It groups all messages by key, which are likely to be something like `errors` and `messags` (@todo is that actually right?), and provides convenience methods for getting all its stored messages or only those for a particular key, and for outputting these messages in various formats.

You could always just spin up a new instance of `MessageBag` manually like in Example 5-12.

Example 5-12. Manually creating and using MessageBag

```
$messages = [
    'errors' => [
        'Somethign went wrong with edit 1!'
    ],
    'messages' => [
        'Edit 2 was successful.'
    ]
];
$messagebag = new \Illuminate\Support\MessageBag($messages);

// Check for errors; if there are any, decorate and echo
if ($messagebag->has('errors')) {
    echo '<ul id="errors">';
    foreach ($messagebag->get('errors', '<li><b>:message</b></li>') as $error) {
        echo $error;
    }
    echo '</ul>';
}
```

Message bags are also closely connected to Laravel's validators: when validators return errors, they actually return an instance of `MessageBag`, which you can then pass to your view or attach to a redirect using `redirect(route)→withErrors($messagebag)`.

Laravel passes an empty instance of `MessageBag` to every view, assigned to the variable `$errors`, and if you've flashed a message bag using `withErrors()` on a redirect, it'll get assigned to that `$errors` variable instead. That means every view can always assume it has an `$errors` MessageBag it can check in whatever place it does its validation, which leads to Example 5-13 as a common snippet developers place on every page.

Example 5-13. Error bag snippet

```
// partials/errors.blade.php
@if ($errors->any())
        <div class="alert alert-danger">
        <ul>
        @foreach ($errors as $error)
            <li>{{ $error }}</li>
        @endforeach
        </ul>
        </div>
@endif
```

Named error bags

Sometimes you need to differentiate message bags not just by key ("notices" vs "errors") but also by component. Maybe you have a login form and a signup form on the same page; how do you differentiate?

When you send errors along a redirect using `withErrors`, the second parameter is the name of the bag: `redirect(dashboard")→withErrors($validator, 'login)`. Then on the dashboard, you can use `$errors→login` to call all of the methods we saw before: `any()`, `count()`, and more.

String helpers, pluralization, and localization

As developers, we tend to look at blocks of text as just so much `Lorem ipsum`, waiting for the client to put real content into it. Seldom are we involved in any logic inside these blocks.

But there are a few circumstances where you'll be grateful for the tools Laravel provides for string manipulation.

The string helpers and pluralization

Laravel has a series of helpers for manipulating strings. They're available as methods on the `Str` class (e.g. `Str::plural()`, but most also have a function shortcut (e.g. `str_plural()`).

The Laravel documentation covers all of the string helpers in detail (TODO INSERT LINK *https://laravel.com/docs/5.1/helpers)*, but here are a few of the most-commonly-used helpers:

- *e*: a shortcut for `html_entities`
- *starts_with, ends_with, str_contains*: check a string (first parametr) to see if it starts with, ends with, or contains another string (second parameter)
- *str_is*: checks whether a string (second parameter) matches a particular pattern (first parameter)—for example, `foo*` will match `foobar` and `foobaz`
- *str_slug*: converts a string to a URL-type slug with hyphens
- *str_plural(word, num), str_singular*: pluralizes a word or singularizes it; English-only

@todo finish this, probably expand the things above

Localization

Localization allows you to define multiple languages and mark any strings as targets for translation. You can set a fallback language, and even handle pluralization variances.

In Laravel, you'll need to set an App locale at some point during the page load so the localization helpers know which bucket of translations to pull from. You'll do this with `App::setLocal($localeName)`, and you can run it in a service provider or a route or wherever else.

You can define your fallback locale in `config/app.php`, where you should find a `fallback_local` key.

@todo Sidebar or whatever about the naming scheme

Basic localization

So, how do we call for a translated string? There's a helper `trans($key)` that will pull the string for the current locale for the passed key, and if it doesn't exist, it'll grab it from the default locale. See Example 5-14 to see how a basic translation works.

Example 5-14. Basic use of `trans()`

```
echo trans('messages.welcome');
```

Let's assume we are using the `en` locale right now. Laravel will look for a file in `resources/lang/es/messages.php`, which it will expect to return an array. It'll look for a `welcome` key on that array, and if it exists, it'll return its value. Take a look at Example 5-15 for a sample.

Example 5-15. Using a translation

```
// resources/lang/en/messages.php
return [
    'welcome' => 'Welcome to our site!'
];

// routes.php
Route::get('/en/welcome', function () {
    App::setLocal('en');
    return view('welcome');
});

// resources/views/welcome.blade.php
{{ trans('messages.welcome') }}
```

Parameters in localization

The above examples are relatively simple. Let's dig into some that are more complex. What if we want to greet the user? Take a look at Example 5-16.

Example 5-16. Parameters in translations

```
// resources/lang/en/messages.php
return [
    'welcome' => 'Welcome back, :name!'
];

// resources/views/welcome.blade.php
{{ trans('messages.welcome', ['name' => 'Jose']) }}
```

As you can see, prepending a word with a colon (`:name`) marks it as a placeholder that can be replaced. The second, optional, parameter of `trans()` is an of values to put into placeholders.

Pluralization in localization

We already covered pluralization above, so now just imagine you're defining your own pluralization rules. There are two ways to do it, so let's start with the simplest, in Example 5-17.

Example 5-17. Defining a simple translation with an option for pluralization

```
// resources/lang/en/messages.php
return [
    'task-deletion' => 'You have deleted a task|You have succesfully deleted tasks'
];

// resources/views/dashboard.blade.php
@if ($numTasksDeleted > 0)
{{ trans_choice('messages.task-deletion', $numTasksDeleted) }}
@endif
```

As you can see, we also have a `trans_choice()` method, which takes the count of items effected as its second parameter, and from this it will determine which string to use.

You can also use any translation definitions that are compatile with Symfony's much more complex Translation component; see Example 5-18 for an example.

Example 5-18. An example of Symfony's Translation component

```
// resources/lang/es/messages.php
return [
    'task-deletion' => "{0} You didn't manage to delete any tasks.|[1,4] You deleted a few tasks.|[5,I
];
```

Testing

Testing with Elixir

You're not going to be writing any tests around your Elixir tasks. However, Elixir provides some functions that will help your testing, so let's talk about that for a second.

@todo talk abotu phpunit

Testing message and error bags

There are two primary ways of testing messages passed along with message and error bags.

First, you can perform a behavior in your application tests that set a message that will eventually be displayed somewhere; then redirect to that page and asser that the appropriate message is showed.

Second, for errors (which is the most common use case), you can assert the session has errors with `$this→assertSessionHasErrors($bindings = [], $format = null)`. Take a look at Example 5-19 to see what this might look like.

Example 5-19. Asserting the session has errors

```
public function test_missing_email_field_errors()
{
    $this->post('person/create', ['name' => 'Japheth']);
    $this->assertSessionHasErrors(['email']);
}
```

Translation and localization

The simplest way to test localization is with application tests. Set the appropriate context (whether by URL or session), `visit()` the page, and assert that you see the appropriate content.

TL;DR

As a full-stack framework, Laravel provides tools and components for the frontend as well as the back.

Elixir is a wrapper around common Gulp build steps that makes it easy to use the most modern build steps simply. Elixir makes it easy to add CSS preprocessors; JavaScript transpilation, concatenation, and minification; and much more.

Laravel also others other internal tools that target the frontend, including pagination, message and error bags, and localization.

CHAPTER 6

Collecting and Handling User Data

Web sites that benefit from a framework like Laravel often don't just serve static content. Many deal with complex and mixed data sources, and one of the most common (and most complex) is user input and its myriad forms: URL paths, query parameters, POSTed data, and file uploads.

Laravel provides a collection of tools for gathering, validating, normalizing, and filtering user-provided data in its many forms.

The Request façade

The most common tool for accessing user data in laravel is the `Request` Façade. It gives easy access to all of the ways users can give input to your site: `POST`, posted JSON, `GET` (query parameters), and URL segments.

> The Request façade actualy exposes the entire Illuminate HTTP request object, but for now we're only going to be looking at its methods which specifically relate to user data.

`Request::all`

Just like the name suggests, `Request::all()` gives you an array containing all of the input the user has provided, from every source. Let's say, for some reason, you decided to have a form `POST` to a URL with a query parameter (e.g. sending a `POST` to *http://myapp.com/post?utm=12345)*. Take a look at Example 6-1 to see what you'd get from `Request::all()`. *Note: Request::all() also contains information about any files that were uploaded, but we'll cover that later in the chapter.*

Example 6-1. `Request::all()`

```
// GET route form view at /get-route
<form method="post" action="/post-route?utm=12345">
    {{ csrf_field() }}
    <input type="text" name="firstName">
    <input type="submit">
</form>

// POST route at /post-route
var_dump(Request::all());

// Outputs:
/**
 * [
 *     '_token' => 'CSRF token here',
 *     'firstName' => 'value',
 *     'utm' => 12345
 * ]
 */
```

`Request::except` and `Request::only`

`Request::except` provides the same output as `Request::all`, but you can choose one or more fields to exclude—for example, `_token`. You can pass it either a string or an array of strings.

Let's look at the same form as in Example 6-1, but using `Request::except`, in Example 6-2.

Example 6-2. `Request::except()`

```
// POST route at /post-route
var_dump(Request::except('_token'));

// Outputs:
/**
 * [
 *     'firstName' => 'value',
 *     'utm' => 12345
 * ]
 */
```

`Request::only` is the inverse of `Request::except`, as you can see in Example 6-3.

Example 6-3. `Request::except()`

```
// POST route at /post-route
var_dump(Request::only(['firstName', 'utm']));
```

```
// Outputs:
/**
 * [
 *     'firstName' => 'value',
 *     'utm' => 12345
 * ]
 */
```

Request::has and Request::exists

With `Request::has` you can detect whether there's a particular piece of user input available to you. Check out Example 6-4 for an analytics example with our `utm` query string parameter from the previous examples.

Example 6-4. `Request::has()`

```
// POST route at /post-route
if (Request::has('utm')) {
    // Do some analytics work
}
```

`Request::exists` is the same as `Request::has`, exist it will return `TRUE` if the key exists but is empty.

Request::input

Where `Request::all`, `Request::except`, and `Request::only` operate on the full array of input provided by the user, `Request::input` allows you to get the value of just a single field. Note that the second parameter is the default value, so if the user hasn't passed in a value, you can have a sensible (and non-breaking) fallback.

Example 6-5. `Request::input()`

```
// POST route at /post-route
$userName = Request::get('name', '(anonymous)');
```

Array input

Laravel also provides convenience helpers for accessing data from array input. Just use the "dot" notation to indicate walking down the inheritance tree.

Example 6-6. Dot notation to access array values in user data

```
// GET route form view at /get-route
<form method="post" action="/post-route">
    {{ csrf_field() }}
    <input type="text" name="employees[0][firstName]">
    <input type="text" name="employees[0][lastName]">
    <input type="text" name="employees[1][firstName]">
    <input type="text" name="employees[1][lastName]">
    <input type="submit">
</form>
```

```
// POST route at /post-route
$employeeZeroFirstName = Request::input('employees.0.firstName');
$allLastNames = Request::input('employees.*.lastName');
$employeeOne = Request::input('employees.1');

// If forms filled out as "Jim" "Smith" "Bob" "Jones":
// $employeeZeroFirstName = 'Jim';
// $allLastNames = ['Smith', 'Jones'];
// $employeeOne = ['firstName' => 'Bob', 'lastName' => 'Jones']
```

JSON input (and `Request::json`)

So far we've covered input from query strings (`GET`) and form submissions (`POST`). But there's another form of user input that's becoming more common with the advent of JavaScript Single-Page-Apps: The JSON request. It's essentially just a `POST` request with the body set to JSON instead of a traditional form `POST`.

Let's take a look at what it might look like to submit some JSON to a Laravel route, and how to use `Request::input` to pull out that data.

Example 6-7. Getting data from JSON with `Request::input`

```
POST /post-route HTTP/1.1
Content-Type: application/json

{"firstName":"Joe","lastName":"Schmoe","spouse":{"firstName":"Jill","lastName":"Schmoe"}}
```

```
// post-route
$firstName = Request::input('firstName');
$spouseFirstname = Request::input('spouse.firstName');
```

Since `Request::input` is smart enough to pull user data from `GET`, `POST`, or JSON, why would we even worry about using `Response::json` to get that data? There are two possible reasons: First, to be more explicit to other programmers on your project about where you're expecting the data to come from. And second, if the `POST` doesnt have the correct `application/json` headers, `Request::input` won't pick it up as JSON, but `Request::json` will.

Façade namespaces, the `request()` global helper, and injecting `$request`

Any time you're using Façades inside of namespaced classes (e.g. controllers), you'll have to add the full Façade path to the import block at the top of your file (e.g. `use Illuminate\Support\Facades\Request`).

Because of this, several of the Façades also have a companion that's a global helper function. Almost all provide two functions: First, if they're run with no parameter, they expose the same syntax as the façade (e.g. `request()->has()` is the same as `Request::has()`), and second, they have a default behavior for when you pass them a parameter (e.g. `request('firstName')` is a shortcut to `request()->input('firstName')`).

Finally, with `Request`, you can also inject an instance of the Request object (learn more in ???) into any controller method or Route Closure. Just typehint `Illuminate\Http\Request` and you can then use all these same methods on that object instead—e.g. `$request->all()` instead of `Request::all()`. Here's what that typehint might look like:

```
Route::post('form', function (Illuminate\Http\Request $request) {
    var_dump($request->all());
});
```

Route data

It might not be the first thing you'd think when you imagine "user data", but the URL is just as much user data as anything else in this chapter.

There are three primary ways you'll get data from the URL: the Request façade, route parameters, and request objects. We'll cover request objects in ???.

From the façade

The `Request` façade (and the `request()` helper) has several methods available to represent the state of the current page's URL, but right now let's look primarily at getting information about the URL segments.

If you're not familiar with the idea of URL segments, each group of characters between `/` in a URL is called a segment. So, *http://www.myapp.com/users/15/* has two segments: `users` and `15`.

As you can probably guess, we have two methods available to us: `Request::segments()` returns an array of all segments, and `Request::segment($segmentId)` allows you to get the value of a single segment. Note that segments are returned on a 1-based index, so in the example above, `Request::segment(1)` would return *`users`*.

The Request façade and helper—and request objects, which we haven't even touched yet—provide quite a few more methods to help you get data out of the URL. To learn more, check out ???.

From route parameters

The other primary way we get data about the URL is from route parameters, which are injected into the controller method or Closure that is serving a current route like in Example 6-8.

Example 6-8. Getting URL details from route parameters

```
// routes.php
Route::get('users/{id}', function ($id) {
    // If the user visits myapp.com/users/15/, $id will equal 15
});
```

To learn more about routes and route binding, check out Chapter 3.

Uploaded files

We've talked about different ways users can input text data, but there's also the matter of file uploads to consider. The Request façade provides access to any uploaded files using the `Request::file` method, which takes the file's input name as a parameter and returns an instance of `Symfony\Component\HttpFoundation\File\UploadedFile`.

Let's walk through an example. First, our form, in Example 6-9.

Example 6-9. A form to upload files

```
<form method="post" enctype="multipart/form-data">
    {{ csrf_field() }}
    <input type="text" name="name">
    <input type="file" name="profile_picture">
    <input type="submit">
</form>
```

Now, let's take a look at what we get from running `Request::all()`, in Example 6-10. Note that `Request::input('profile_picture')` will return `null`; we need to use `Request::file('profile_picture')` instead.

Example 6-10. The output from submitting Example 6-9

```
// In controller/route Closure
var_dump(Request::all());

// Output:
// [
//     "_token" => "token here"
//     "name" => "asdf"
//     "profile_picture" => UploadedFile {}
// ]

if (Request::hasFile('profile_picture')) {
    var_dump(Request::file('profile_picture'));
}

// Output:
// UploadedFile (details)
```

Validating a file upload

As you can see in Example 6-10, we have access to `Request::hasFile` to see whether the user uploaded a file. We can also check whether the file upload was successful using `isValid` on the file itself:

```
if (Request::file('profile_picture')->isValid()) {
    //
}
```

Because `isValid` is called on the file itself, it will error if the user didn't upload a file. So, to check for both, you'd need to check for the file's existence first:

```
if (
    Request::hasFile('profile_picture') &&
    Request::file('profile_picture')->isValid()
    ) {
    //
}
```

The `UploadedFile` class extends PHP's native `SplFileInfo` with methods allowing you to easily inspect and manipulate the file. This list isn't exhaustive, but can give you a taste of what you can do:

- `guessExtension()`
- `getMimeType()`
- `move($directory, $newName = null)`
- `getClientOriginalName()`

- `getClientOriginalExtension()`
- `getClientMimeType()`
- `guessClientExtension()`
- `getClientSize()`
- `getError()`
- `isValid()`

As you can see, most of the methods have to do with getting information about the uploaded file, but there's one that you'll likely use more than all the others: `move()`. You can see a common workflow in Example 6-11.

Example 6-11. Common file upload workflow

```
if (Request::hasFile('profile_picture')) {
    $file = Request::file('profile_picture');
    if (! $file->isValid()) {
        // handle invalid state; likely redirect with an error message
    }

    $newFileName = Str::random(32) . '.' . $file->guessExtension();
    $file->move('profile_picture_path_here', $newFileName);
    Auth::user()->profile_picture = $newFileName;
    Auth::user()->save();
}
```

If you get `null` when you run `Request::file`, you might've forgotten to set the encoding type on your form. Make sure to add the property `enctype="multipart/form-data"` on your form.

```
<form method="post" enctype="multipart/form-data">
```

Validation

Laravel has quite a few ways you can validate incoming data. We'll cover Form Requests in the next section, so that leaves us with two primary options: Manual and using `validate()` in the controller. Let's start with the simpler, and more common, `validate()`.

`validate()` in the controller using `ValidatesRequests`

Out of the box, all Laravel controllers use the `ValidatesRequests` trait, which provides a convenient `validate()` method. Let's take a look at what it looks like in Example 6-12.

Example 6-12. Basic usage of controller validation

```php
// app/Http/routes.php
<?php
Route::get('recipes/create', 'RecipesController@create');
Route::post('recipes', 'RecipesController@store');
```

```php
// app/Http/Controllers/RecipesController.php
<?php

namespace App\Http\Controllers;

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

class RecipesController extends Controller
{
    public function create()
    {
        return view ('recipes.create');
    }

    public function store(Request $request)
    {
        $this->validate($request, [
            'title' => 'required|unique:recipes|max:125',
            'body' => 'required'
        ]);

        // Recipe is valid; proceed to save it
    }
}
```

We only have four lines of code running our validation here, but they're doing a lot.

First, we're explicitly defining the fields we expect and applying rules (here separated by | pipes) to each individually.

Next, the `validate` method checks the incoming data from the `$request` (which means it can use `$request→all()` or `$request→get()` just like we learned about earlier in the chapter) and determines whether or not it is valid.

If the data is valid, the `validate` method ends and you can move on with your controller method, saving the data or whatever else.

But if the data isn't valid, it throws a `ValidationException`. This contains instructions to the router about how to handle this exception. If the request is AJAX (or if it's requesting JSON as a response), the exception will create a JSON response containing the validation errors. If not, the exception will return a redirect to the previous page, together with all of the user input and the validation errors—perfect for repopulating a failed form and showing some errors.

More on Laravel's validation rules

In our examples here (and in the docs) we're using the "pipe" syntax: `'fieldname': 'rule|otherRule|anotherRule'`. But you can also use the array syntax to do the same thing: `'fieldname': ['rule', 'otherRule', 'anotherRule']`.

There's also the option for validating nested properties. These matter if you use HTML's array syntax, which allows you to, for example, have multiple "users" on an HTML form, each of which have a name and email address. Here's how you validate that:

```
$this->validate($request, [
    'user.name' => 'required',
    'user.email' => 'required|email',
]);
```

Finally, we don't have enough space to cover every possible validation rule here, but here are a few of the most common rules:

- required
- email
- alpha
- alpha dash
- alpha numeric
- between (date)
- exists (database)
- integer
- min
- max
- required if
- required unless
- same
- size
- unique (database)

Manual validation

If you are not working in a controller, or if you have some other reason that the above flow is not a good fit, you can manually create a Validator instance and check for success or failure like in Example 6-13.

Example 6-13. Manual validation

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

Route::post('recipes', function (Illuminate\Http\Request $request) {
    $validator = Validator::make($request->all(), [
        'title' => 'required|unique:recipes|max:125',
        'body' => 'required'
    ]);

    if ($validator->fails()) {
        return redirect('recipes/create')
            ->withErrors($validator)
            ->withInput();
    }

    // Recipe is valid; proceed to save it
});
```

As you can see, we create an instance of a validator by passing it our input as the first parameter and the validation rules as the second parameter. The validator exposes a `fails()` method that we can check against, and also can be passed into the `withErrors()` method of the redirect.

Displaying validation error messages

We've already covered much of this in Chapter 5, but here's a quick refresher on how to display errors from validation.

The `validate()` method in controllers (and the `withErrors()` method on redirects that it relies on) flash any errors to the session. These errors are made available to the view you're being redirected to as the `$errors` variable. And remember, as a part of Laravel's magic, that `$errors` variable will be available every time you load the view, even if it's just empty, so you don't have to check if it `isset`.

That means you can do something like Example 6-14 on every page.

Example 6-14. Echo validation errors

```
@if (count($errors) > 0)
    <ul id="errors">
        @foreach ($errors->all() as $error)
            <li>{{ $error }}</li>
        @endforeach
    </ul>
@endif
```

Form Requests

If you find yourself wishing you could extract the validation, authentication, and redirection aspects of your controller code, there's a structure available for you called the Form Request. Each Form Request will usually explicitly map to a single HTTP request—e.g. "Create Comment".

Creating a Form Request

You can create a new Form Request using Artisan:

```
php artisan make:request CreateCommentRequest
```

You now have a Form Request object available at `app/Http/Requests/CreateCommentRequest.php`.

Every Form Request class provides either one or two public methods. The first is `rules()`, which needs to return an array of validation rules for this request. And the second (optional) method is `authorize()`; if this returns true, the user is authorized to perform this request, and if false, the user is rejected.

Take a look at Example 6-15 to see a sample form request.

Example 6-15. Sample Form Request

```
<?php

namespace App\Http\Requests;

use App\Http\Requests\Request;

class CreateCommentRequest extends Request
{
    public function rules()
    {
        return [
            'body' => 'required|max:1000'
        ];
    }

    public function authorize()
    {
        $blogPostId = $this->route('blogPost');

        return BlogPost::where('id', $blogPostId)
            ->where('user_id', Auth::user()->id)->exists();
    }
}
```

The `rules()` section of Example 6-15 is pretty self-explanatory, but let's look at `authorize()` briefly.

We're grabbing the segment from the route named `blogPost`. That's implying the route definition for this route probably looks a bit like this: `Route::post('blogPosts/{blogPost}\', function () { // Do stuff })`. As you can see, we named the route parameter `blogPost`, which makes it accessible in our Request using `$this→route('parameter name')`.

We then look whether any blog posts exist with that identifier that are owned by the currently-logged-in user.

Using a Form Request

Now that we've create a Form Request object, how do we use it? It's a little bit of Laravel magic. Any route (Closure or controller method) that typehints a Form Request as one of its parameters will benefit from the definitions of that Form Request.

So, let's try it out, in Example 6-16.

Example 6-16. Using a Form Request

```
Route::post('comments', function (\App\Http\Requests\CreateCommentRequest $request) {
    // Store comment
});
```

You might be wondering where we call the Form Request, but Laravel does it for us. It validates the user input and authorizes their request. If the input is invalid, it'll act just like the in-controller validate method works, redirecting them to the previous page with their input preserved and with the appropriate error messages passed along. And if the user is not authenticated, Laravel will return a 403 (Forbidden) error and not execute the route code.

Eloquent model mass assignment

It's a common pattern to pass the entirety of a form's input directly to a database model. In Laravel, that might look like Example 6-17.

Example 6-17. Passing the entirety of a form to an Eloquent model

```
Route::post('posts', function () {
    $newPost = Post::create(Request::all());
});
```

We're assuming here that the end user is kind and not malicious, and has purely kept only the fields we want them to edit. Maybe the post `title` or `body`.

But what if our end user can guess, or discern, that we have a `author_id` field on that `posts` table? What if they used their browser tools to add an `author_id` field and set the ID to be someone else's ID, and then impersonated them by creating fake blog posts attributed to that other person?

Eloquent has a concept called "mass assignment", which allows you to either whitelist fields that are fillable in this way (using the model's `$fillable` property) or blacklist fields that aren't fillable (using the model's `$guarded` property). In our example, we might want to fill it out like Example 6-18 to keep our app safe.

Example 6-18. Guarding an Eloquent model from mischevious mass assignment

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    protected $guarded = ['author_id'];
}
```

{{ vs {!!

Any time you display content on a web page that was created by a user, you need to guard from script injection.

Let's say you allow your users to write blog posts on your site.

You probably don't want them to be able to inject malicious JavaScript that's injected and run in your unsuspecting visitors' browsers, right? So you'll want to escape any user input that you show on the page to avoid this.

Thankfully, it's actually already almost covered for you. If you use Laravel's Blade templating engine, the default "echo" syntax (`{{ $stuffToEcho }}`) already runs the output through `htmlentities()` (PHP's best way of making user content safe to echo) automatically. You actually have to do **extra** work to get out of it, by using the `{!! $stuffToEcho !!}` syntax.

Testing

If you're interested in testing your interactions with user input, you're probably most interested in simulating valid and invalid user input and ensuring that, if the input is invalid, the user is redirected, and if the input is valid, it ends up in the proper place (e.g. the database).

Laravel's end-to-end application testing makes this simple. Let's start with an invalid route that we expect to be rejected, in Example 6-19.

Example 6-19. Testing that invalid input should be rejected

```
public function test_input_missing_a_title_is_rejected()
{
    $this->post('posts', ['body' => 'This is the body of my post']);
    $this->assertRedirectedTo('posts/create');
    $this->assertSessionHasErrors();
    $this->assertHasOldInput();
}
```

We've asserted that, after invalid input, the user was redirected, with errors, and with the old input correctly passed back. You can see we're using a few custom PHPUnit assertions that Laravel adds here.

So, how do we test our route's success? Check out Example 6-20.

Example 6-20. Testing that valid input should be processed

```
public function test_valid_input_should_create_a_post_in_the_database()
{
    $this->post('posts', ['title' => 'Post Title', 'body' => 'This is the body']);
    $this->seeInDatabase(['title' => 'Post Title']);
}
```

Note that, if you're testing something using the database, you'll need to learn more about Database Migrations and Transactions. More on that in ???.

TL;DR

There are a lot of ways to get the same data: The Request façade, the `request()` global helper, and injecting an instance of `Illuminate\Http\Request`. Each expose the ability to get all input, some input, or specific pieces of data, and files and JSON input can have some special considerations at times.

URI segments are also a possible source for user-inputted data, and they're also accessible via the Request tools. Files are as well.

Validation can be performed manually with `Validator::make`, or automatically using controllers' `$this->validate()` or by using Form Requests. Each automatic tool, upon invalidation, redirects the user to the previous page with all old input stored and errors passed along.

Views and Eloquent models also need to be protected from nefarious user input.