BY TONY LEA & THE DEVDOJO

EDITION 1

THE LARAVEL SURVIVAL GUIDE

A FUN WAY TO LEARN THE BASICS OF LARAVEL
& SAVE YOURSELF FROM BECOMING A ZOMBIE DEVELOPER!

The Laravel Survival Guide

Tony Lea

Table of Contents

Thanks

I want to give a special thanks to my family for always being supportive. I especially want to thank my wife for being an amazing woman and for pushing me to try and be the best person I can be. This book would not be possible without the love and support of my amazing family.

Additionally, I want to thank to David Cullinan for helping out with book.

Ok, enough with all the thanks. Let’s start learning Laravel!

Introduction

Why The Book? Well, it's not really a book… It's more of a guide (hence the title). A guide to save yourself and others from becoming a zombie developer!

What exactly is a zombie developer? Well, a zombie developer is a developer like you or I, yet they mindlessly hack on PHP apps and do the same thing over and over. These repetitive tasks are incredibly time consuming, and make the developer brain dead. When this happens it gives them a thirst for blood and an urge to kill.

So, instead of letting this happen the developer could have used the amazing Laravel framework for rapid application development. This will help them keep their sanity and it will make coding enjoyable again. Oh, yeah… And it'll save lives.

By learning the basics of Laravel you can save yourself, and possibly others, from becoming a deteriorating zombie developer.

Don't let that inner zombie revive, be sure to keep in hand this Laravel survival guide.

Chapter 1 - Getting Started

A zombie developer is very slow at setting up a new project, whereas a Laravel developer can run a few commands to set up a new project in a matter of seconds!

Being aware of one's environment is crucial to surviving the zombie developer apocalypse. In this chapter, we will briefly go over setting up your local environment.

> Quick note: We will not go into full detail of installing system requirements such as PHP, MySQL, and Apache. Instead, we are just going to give a quick overview on setting up your local development environment so that way we can start getting into the code as soon as possible.

Your Local Development Environment

A local environment, also referred to as the “development environment”, is when you work on your web app from your computer. When you are ready to make your application available for the world to see you will move your code to another server referred to as the “production environment”.

So, when we set up your local development environment we are referring to setting up the required applications to run a Laravel app on your computer.

A basic local development environment for Laravel typically requires 3 applications, which are:

1. Apache or Nginx (the **web server** for your application)
2. MySQL (the **database** for your application)
3. PHP (the **server-side scripting** language for your application)

There are many programs for each operating system that will install all these applications for you. Just to name a few there is MAMP, WampServer, and XAMPP. You can feel free to check out the links below for these applications:

- https://www.mamp.info/en/ (Mac and Windows)
- http://www.wampserver.com/en/ (Windows)
- https://www.apachefriends.org/ (Mac, Windows, and Linux)

Be sure to read the docs for all the system requirements at http://laravel.com/docs.

There is also one last way of getting your local development environment setup for Laravel that is well worth mentioning. It is called a virtual machine that has all the system requirements already installed. You can learn more about setting up a Laravel virtual machine at http://laravel.com/docs/homestead.

There is no right or wrong way to setup your local development environment as long as you meet the minimum system requirements for Laravel. Find a way that works for you and start building the next latest and greatest web app!

Okay, Let’s move on to Composer and the Laravel Installer.

Chapter 2 - Composer & The Laravel Installer

A zombie developer manually moves files into their project, whereas a Laravel developer leverages composer to install tools and libraries.

Taking on the zombie developer apocalypse on your own would be almost impossible. Getting help from others is essential, and that's exactly what composer allows us to do. Composer is used to include libraries from other developers into our application.

Let's continue.

Composer & The Laravel Installer

How easy would it be if you could open up a command prompt and type in:

```
$ laravel new blog
```

And a Laravel app is created inside a folder named "blog". Well, that's what the laravel installer does. So, let's learn how we can use this on our computer.

The Laravel installer, as well as many PHP packages, make use of a dependency manager called Composer (http://getcomposer.org) to add this functionality.

So, What exactly is a dependency manager?

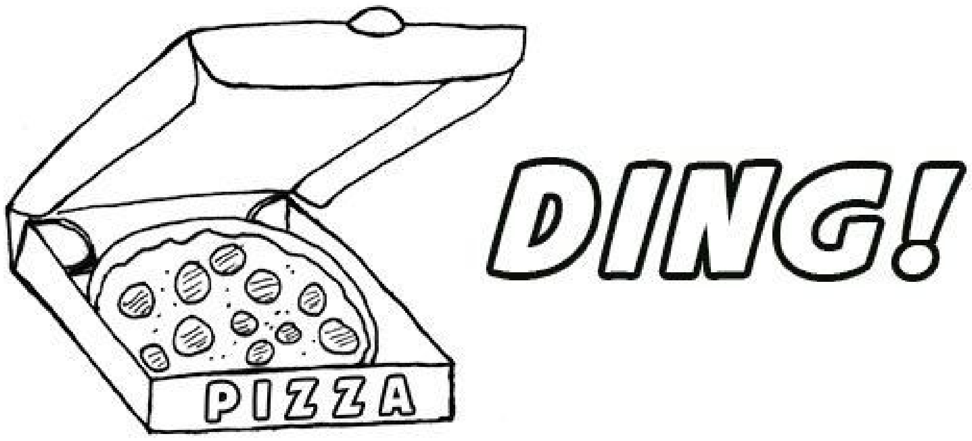
Well, a dependency manager is nothing more than a tool to manage your dependencies.

"WHAT!!!?", yeah the definition still sounds pretty abstract, right?

Let me put this another way to help you understand how Composer works.

I know you are probably a fan of eating pizza instead of eating brains, so let's pretend we could use a command to make us a pizza:

```
$ composer make pizza
```



By default, we are given a pepperoni pizza. But let’s say we wanted this command to make us a pizza with different toppings. Perhaps we wanted a meat-lovers pizza. We would probably need the following toppings:

```
{
    "toppings" : [
        "peperoni", "ham", "bacon", "beef", "sausage"
    ]
}
```

Now, if we save this file in our current directory and name it ‘composer.json’ and run the command again:

```
$ composer make pizza
```

DING! We now get our meat-lovers pizza instead of our pepperoni pizza.

Hazzzaa!

As you can see Composer is a way of managing the things we need to build our app (or pizza).

Composer is also a command line tool we can use it to install other command line tools.

One of those tools is the Laravel installer.

To add the Laravel installer to our computer, we must first install composer.

Visit https://getcomposer.org/, click on the ‘Getting Started’ button, navigate to ‘Installing Globally’, and walk through how to globally install composer on your machine.

After downloading and installing composer you can run the following command to add the Laravel installer.

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

Now we have successfully added the laravel installer to your machine, and you can easily create a new laravel app by typing:

```
$ laravel new app_name
```

Then navigate to your new `app_name` folder in a command prompt. And run:

```
$ php artisan serve
```

Finally, navigate to http://localhost:8000/ in your browser and you will see a ‘Welcome to Laravel’ screen.

Now you’re ready to start building your amazing app!

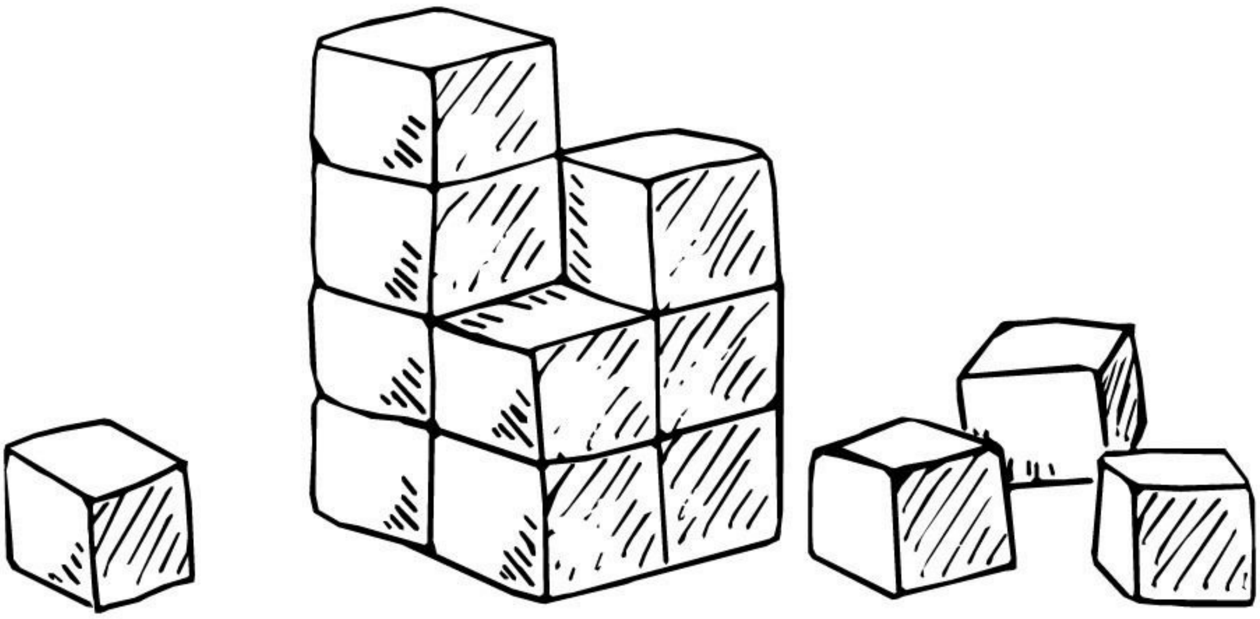
Warning: If the laravel installer does not globally work you may need to specify where the composer bin directory is located on your machine. Visit http://devdojo.com/post/composer-bin-directory-path to learn how to do this.

How awesome is this?

With a single command line, you can be up and running with a new Laravel app in a few seconds!

You can learn more about the Laravel installer at http://laravel.com/docs.

Chapter 3 - The Laravel Structure



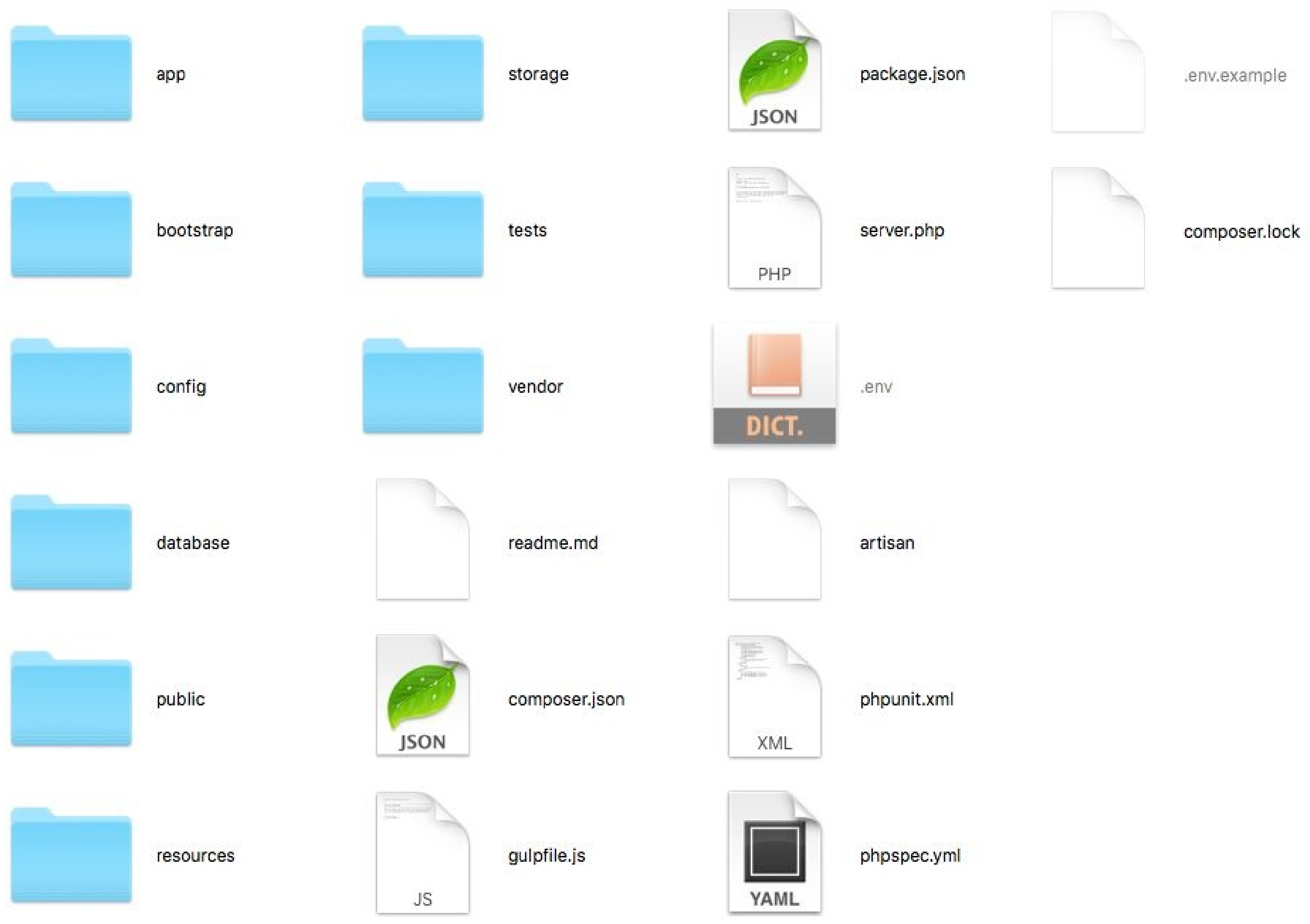
A zombie developer puts files all over the place causing an unstable structure, whereas a Laravel developer keeps their structure clean and consistent.

A solid structure is essential for making your app efficient and awesome. Zombie developers are used to ruins and destruction, but as a new Laravel developer, you will get accustomed to a solid foundation and structure.

In this chapter, we are going to give you a brief overview of the Laravel file structure.

The Laravel Structure

In a new laravel project you will have the following code structure:



You will see nine folders which are:

1. app
2. bootstrap
3. config
4. database
5. public
6. resources
7. storage
8. tests
9. vendor

I'm not going to go into detail with all the files; however, I will give you a brief rundown of each folder.

App

This is the directory that has all our application logic. In this folder, we will put all our models, controllers, services, and many other classes.

Bootstrap

This folder is used to bootstrap laravel (startup laravel).

Config

This file will contain many of our global configurations for our application.

Database

This folder contains our database files such as migrations and seeds.

Public

This public folder contains many of the applications assets such as images, stylesheets, and scripts.

Resources

We will put our view files in this folder. The views are the pages that a user sees.

Storage

Laravel uses this folder to store sessions, caches, and logs in this folder.

Test

This folder contains files that we use to test the logic of our application.

Vendor

This is the folder that contains our dependencies. When you add new libraries (toppings) to your app, this is the folder that will contain those libraries.

Do you recognize the composer.json file from the image above? Remember this is where we define our dependencies (pizza toppings) for our app.

One important file worth mentioning is a file called .env; this is the file that contains configurations such as debug mode and database credentials. So, when you need to connect a database to your Laravel app you will need to update the following code in that file:

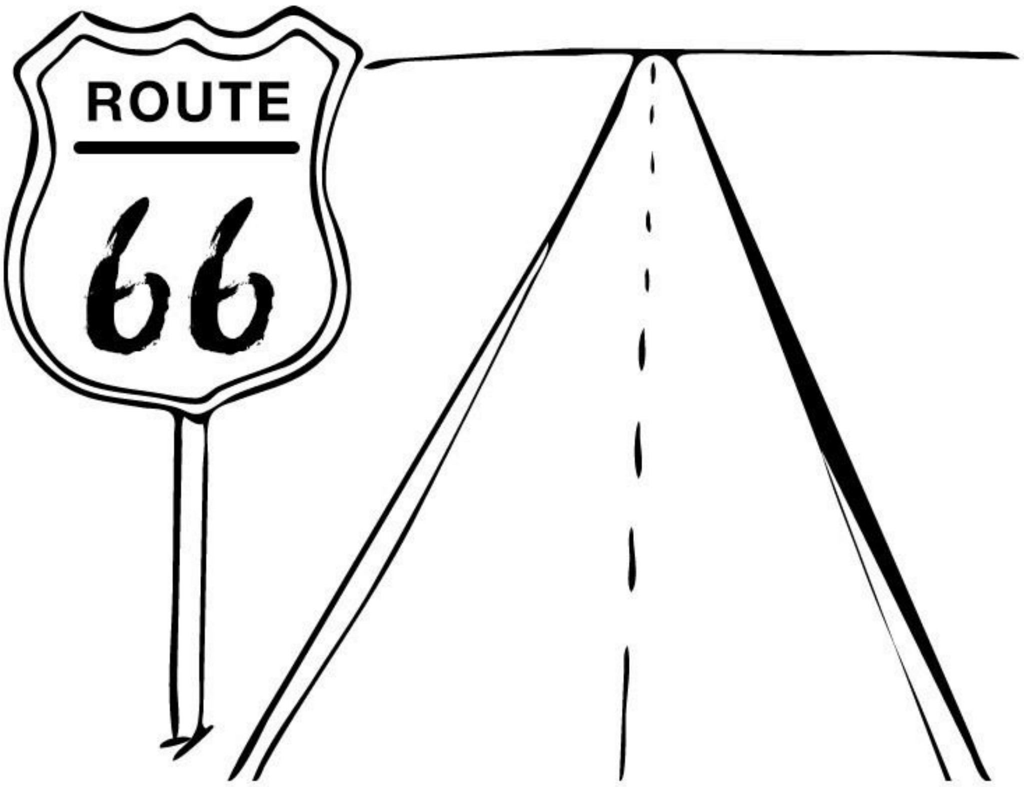
```
DB_HOST=localhost
DB_DATABASE=homestead
DB_USERNAME=homestead
DB_PASSWORD=secret
```

You will replace the database name, username, and password accordingly based on your credentials. Add your database credentials to these variables and it will be available for your app to use.

Finally, there is one particular file that I want to point out. The *routes.php* file, which is located in *app/Http/routes.php*. This is the file that we will add all of the routes for our application, and this is what we are going to cover in the next chapter.

So, Let’s move onto some fun stuff!

Chapter 4 - Routing



A zombie developer reinvents the wheel and creates their own routing system, whereas a Laravel developer leverages the built-in router that is simple to use, extremely flexible, and super efficient.

Using a powerful routing system in your app is crucial for keeping your sanity and preventing brain deterioration. When a user navigates your app they won't run into a dead end; they'll hit the correct "route" instead.

Routing overview

Just to be sure everyone is on the same page, we’ll give you a brief run down of app routing.

You can think of a route as being similar to a road. For instance, “We drove down the road (route) to the graveyard.” When referring to routes in an application, it is essentially the same. When you type a website URL like `site.com/graveyard`, you are telling the browser that the graveyard is the route you want to take. The application then says, “Ok, you want to go to the ‘graveyard’? This is the output I have for the graveyard route.”

This can be done very easily using Laravel, for instance:

```php
<?php

Route::get('graveyard', function(){
     echo 'Welcome to the graveyard!';
});
```

The code above states that when the browser says ‘get’ the ‘graveyard’ route, our app will perform the following function. In the code above our page will display ‘Welcome to the graveyard!’.

Couldn’t be easier, right?

Let’s learn more about the Laravel routing service.

Routing in Laravel

The Laravel route file is located at `app\Http\routes.php`. This is where we will be adding all our routes for our application.

There are 4 basic types of routes we can add. These types are `POST, GET, PUT, and DELETE`, and look like the following:

```php
<?php

Route::post('/zombie', function () {
     echo "We want to create a new zombie";
});

Route::get('/zombie', function () {
     echo 'We want to read or view a zombie';
});

Route::put('/zombie', function () {
     echo "We want to update an existing zombie";
});

Route::delete('/zombie', function () {
     echo "We want to destroy a zombie";
});
```

The POST, GET, PUT, and DELETE methods are all part of a RESTful architecture, where each verb corresponds to an action.

POST

When we POST data to a page, we *Create* an item.

GET

When we GET data from a page, we *Read* an item or a list of items.

PUT

When we PUT data into a page, we *Update* an item.

DELETE

Finally, when we DELETE data from a page, we *DELETE* an item.

This technique is also referred to as CRUD (Create, Read, Update, Delete).

Most often we will use the GET method, but there is also a route we can use to capture any method:

```php
<?php

Route::any('/zombie', function () {
     echo "Any request from this zombie route";
});
```

Awesome!

Ok, how might we access our routes from a browser? Well, like we said, the GET request is usually the method we will use for most requests. If you typed in site.com/zombie we would be directed to the GET method. But, how would we POST data to a route?

Simple enough! We can create a form in HTML that looks like the following:

```html
<form method="POST" action="/zombie">
     ...
     <input type="submit">
</form>
```

When the user submits the form in the code above the data inside the form will be posted to the site.com/zombie POST route.

How about the PUT and DELETE method? Well, the next 2 methods will need to have a hidden input type to specify that it is a PUT or a DELETE. For example:

```html
<!-- PUT METHOD -->
<form method="POST" action="/zombie">
     ...
     <input type="hidden" name="_method" value="PUT">
     <input type="submit">
</form>

<!-- DELETE METHOD -->
<form method="POST" action="/zombie">
     ...
     <input type="hidden" name="_method" value="DELETE">
     <input type="submit">
</form>
```

In the previous code sample the forms will send data to the PUT & DELETE route respectively.

Quick Routing example

Let’s go through a quick route example of how we might destroy a zombie!

First we need to have a form where we can delete a zombie. Let’s say that we have the following code in one of our views:

```html
<form method="POST" action="/zombie">
     <input type="hidden" name="id" value="2">
     <input type="hidden" name="_method" value="DELETE">
     <input type="submit" value="Destroy">
</form>
```

The view above will show a button titled ‘Destroy’. For simplicity purposes we are just hard-coding an input with an ID of 2, but normally this would change based on the `ID` of the zombie you actually want to delete.

Next we need to create our route that will delete that zombie:

```php
<?php

use Illuminate\Http\Request;

Route::delete('/zombie', function(Request $request){
     $id = $request->input('id');
     Zombie::destroy($id);
});
```

And just like that we have destroyed the zombie with an `ID` of 2 :) Pretty cool!

Additionally, notice that we specified a `Request` variable in the function above. This is simply a class provided by Laravel that allows us to capture request information. But, before we can use the Request class we have to specify the namespace:

```
use Illuminate\Http\Request;
```

Quick Note: Unfortunately, the above example will not fully work since we have not setup our database or models just yet, but we will do that in the next chapter.

In the above route examples, we are using *route closures*. Let’s move on to talk about *route closures* vs *route controllers*.

Route Closures vs Route Controller Actions

A route closure is when we specify a route, and we run a function containing code. This is a route closure:

```php
<?php

Route::get('/zombie', function(){
     echo 'Welcome to the Zombie Page!';
});
```

Whereas, with a *route controller action* we run a controller method for a specific route. This is a *route controller action*:

```php
<?php

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

When using a *route controller action*, a method will be run when the route is accessed.

We will talk more about Controllers in the next couple chapters. Just make sure to keep this in mind and it will all start coming together :)

Route Parameters

There may be times that you want to pass a few parameters in your routes. As an example, let’s say that we have a zombie with an ID of 5, and we want to view that zombie information by visiting site.com/zombie/5. We could easily do that with the following route:

```php
<?php

Route::get('/zombie/{id}', function($id){
     echo 'This zombie has an id of' . $id;
});
```

The above would print out “This zombie has an id of 5”.

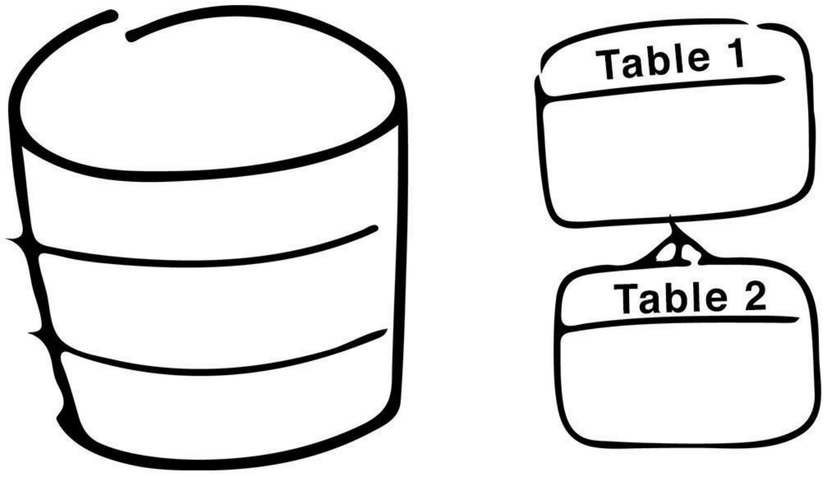
So, if we had our models and database all set up we could get the zombie with an ID of 5 and print out his info, which would look like the following:

```php
<?php

Route::get('/zombie/{id}', function($id){
     $zombie = Zombie::find($id);
     echo 'Name: ' . $zombie->name . '<br />';
     echo 'Strength: ' . $zombie->strength . '<br />';
     echo 'Health: ' . $zombie->health . '<br />';
});
```

Above we are getting the zombie with an ID 5 and we are displaying their name, strength, and their health level.

> The example above will not entirely work since we have not set up our Models or our database yet. So, with that being said let’s move on to talk about our model classes.

Chapter 5 - Models



A zombie developer creates complex SQL queries, whereas a Laravel developer extends the Eloquent Model Class and benefits from beautiful and readable database interaction.

Zombie Developers often use complicated queries that can lead to bad and infectious code. As a Laravel developer we must keep our queries strong & healthy.

We must *MODEL* some good behavior.

Models

A model is a PHP class that handles all the interaction between the code and the database. When using Laravel we can extend the Eloquent Model class and all our interaction will automatically be built-in.

Take a look at what an example `zombie model` would look like (this file would be placed inside of the /app folder):

```php
<?php namespace App;

use Illuminate\Database\Eloquent\Model;

class Zombie extends Model {

    protected $table = 'zombies';

}
```

In the code above we are saying that the class **Zombie** extends from the Illuminate Eloquent Model. And we are saying that our database table name is **zombies**. So, to map the zombie model with the database, we create a simple zombies table in our database that looks like the following:

zombies

Field	Type	Length
id	INT	11
name	VARCHAR	50
strength	VARCHAR	20
health	INT	3
created_at	TIMESTAMP	
updated_at	TIMESTAMP	

In the zombies table we have a unique ID, the name of the zombie, their strength, and their health. Let’s assume that we have this table in our database. In a future chapter, we will be talking about migrations, which allow us to easily create database tables in our code.

After we have a table like the one above, we are free to interact with the database using the Eloquent Model class.

Quick Note: Before interacting with the database you will need to add your database credentials to your `.env` file (briefly covered in chapter 3).

Let’s move on to learning more about this Eloquent Model class.

Eloquent

Models in Laravel extend from the Eloquent class that make your database interactions as clean and easy to use as possible. Eloquent is appropriately named, because, that's exactly how it feels to interact with the database, "Very eloquent." You might remember this code from the previous chapter:

```php
<?php

use App\Zombie as Zombie;

Route::get('/zombie/{id}', function($id){
     $zombie = Zombie::find($id);
     echo 'Name: ' . $zombie->name . '<br />';
     echo 'Strength: ' . $zombie->strength . '<br />';
     echo 'Health: ' . $zombie->health . '<br />';
});
```

Before we would not be able to run our application because our code would not know where to access the *Zombie* class, but now that the Model is created we can access it.

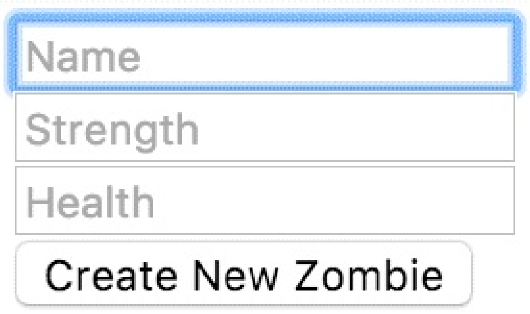
> Note that we are telling our app that when we call Zombie we want to use the zombie class located at App\Zombie. This is referred to as namespaces, something that we will dig further into in a future chapter.

We still have a problem, though.

We will not be able to access the route from above because we do not have any zombies in our database, so let's go ahead and create a new zombie with the following route.

```php
<?php

Route::get('/admin/zombies/create', function(){
     echo '<form method="POST" action="/admin/zombies/create">
               <input type="text" name="name" placeholder="Name"><br>
               <input type="text" name="strength" placeholder="Strength"><br>
               <input type="text" name="health" placeholder="Health"><br>
               <input type="submit" value="Create New Zombie">
          </form>';
});
```

And if we visited that route in our browser (site.com/admin/zombies/create), we would end up with a simple form.



When this form gets submitted it will post to the site.com/admin/zombies/create POST route, which should look like the following:

```php
<?php

Route::post('/admin/zombies/create', function () {
     // create a new zombie
});
```

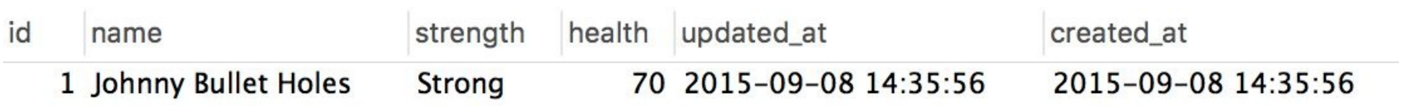
So, if we added the following functionality:

```php
<?php

use App\Zombie as Zombie;
use Illuminate\Http\Request;

Route::post('/admin/zombies/create', function(Request $request){
   // get all the data that has been posted from the form
   $post_data = $request->all();

   // create a new zombie
   $zombie = new Zombie();
   $zombie->name = $post_data['name'];
   $zombie->strength = $post_data['strength'];
   $zombie->health = $post_data['health'];
   $zombie->save();
});
```

And we submitted the form with the following data:

- Name: Johnny Bullet Holes
- Strength: Strong
- Health: 70

Then we would end up with the following row in our database.

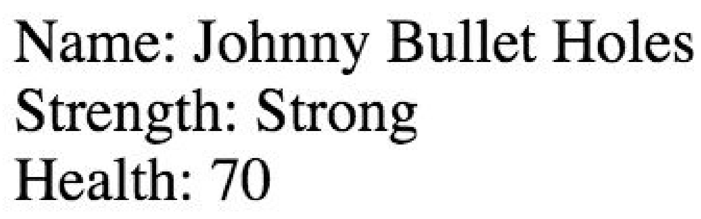


id	name	strength	health	updated_at	created_at
1	Johnny Bullet Holes	Strong	70	2015-09-08 14:35:56	2015-09-08 14:35:56

How easy is that! We just created our first Zombie. So, if we were to visit the following route (site.com/zombie/1), we would be directed to the following route from above:

```php
Route::get('/zombie/{id}', function($id){
     $zombie = Zombie::find($id);
     echo 'Name: ' . $zombie->name . '<br />';
     echo 'Strength: ' . $zombie->strength . '<br />';
     echo 'Health: ' . $zombie->health . '<br />';
});
```

And we would see the following output in our browser.


Name: Johnny Bullet Holes
Strength: Strong
Health: 70

Awesome, right? How much easier could it be? Well, it gets a little easier, instead of creating a zombie by adding the name, strength, and health manually we could always do this in one line.

Check out the following:

```php
<?php

use App\Zombie as Zombie;
use Illuminate\Http\Request;

Route::post('/admin/zombies/create', function(Request $request){
  // get all the data that has been posted from the form
  $post_data = $request->all();

  // create a new zombie
  $zombie = Zombie::create($post_data);
});
?>
```

If we tried to submit to the route above, we would probably get an error message saying 'MassAssignmentException'. This means that we are trying to assign a mass amount of data to our Zombie class, but we have not specified what is ok to add when creating a new Zombie.

This is a level of security that Laravel offers.

To tell our zombie class that we want to be able to create a zombie and allow name, strength, and health all at once we would add the following in our Zombie class:

```php
protected $fillable = ['name', 'strength', 'health'];
```

So, the full class would look like:

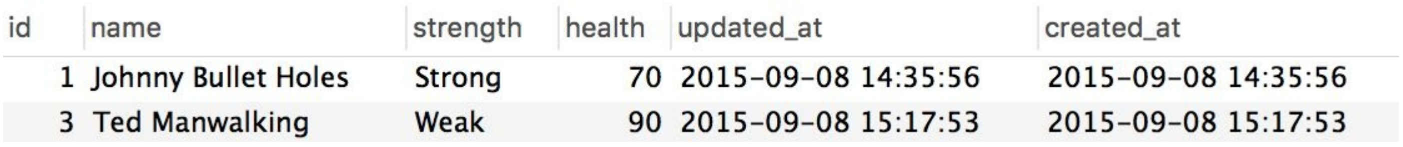
```php
<?php namespace App;

use Illuminate\Database\Eloquent\Model;

class Zombie extends Model {

     protected $table = 'zombies';
     protected $fillable = ['name', 'strength', 'health'];

}
```

And now, if we submitted our form with the new create route above we would not get that mass assignment error. Instead, we would have successfully created another new Zombie.

Let's say that we created another new zombie with the following data:

- Name: Ted Manwalking
- Strength: Weak
- Health: 90

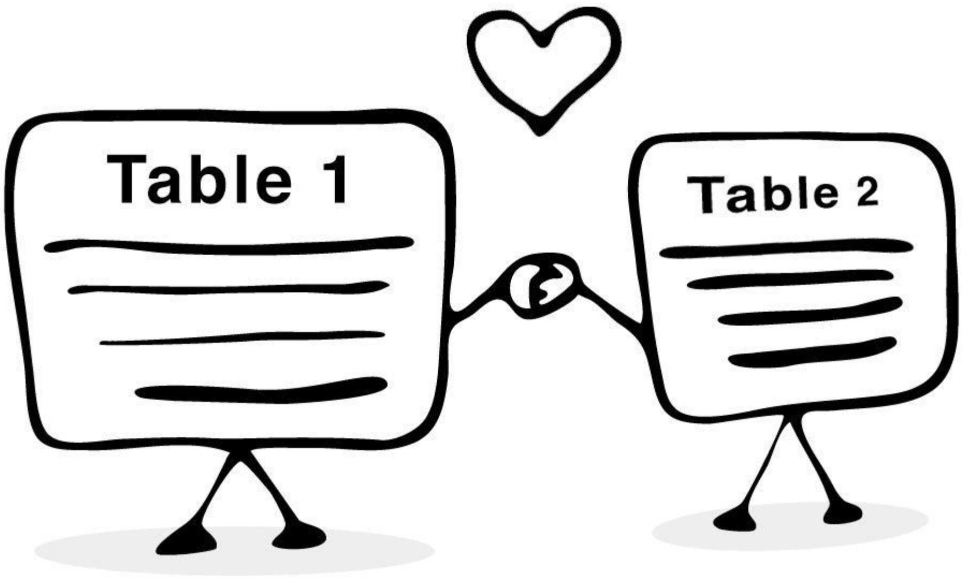
We would now have the following two rows in our database:



id	name	strength	health	updated_at	created_at
1	Johnny Bullet Holes	Strong	70	2015-09-08 14:35:56	2015-09-08 14:35:56
3	Ted Manwalking	Weak	90	2015-09-08 15:17:53	2015-09-08 15:17:53

Using Eloquent makes it super easy to create, read, update, and delete data from our database. Let's move onto relationships, which allow us to easily bind data between tables in a database.

Chapter 6 - Model Relationships



A zombie developer is not very good at relationships, whereas a Laravel developer is great at implementing database relationships.

Zombies lack the intellegence to build meaningful relationships.

If they were to use Laravel’s Eloquent class they could create easy to use relationships between tables.

Model Relationships

Relationships are a way of binding data between tables. Let's say for instance you have a blog that has a 'posts' and a 'comments' table.

These two tables probably have a relationship. As an example, a POST probably HAS MANY COMMENTS, and vice versa a COMMENT probably BELONGS TO a POST. The relationship between the POST to COMMENTS is a HAS MANY relationship, and the relationship between the COMMENTS and POSTS is a BELONGS TO relationship.

Using our Zombie table from above we are also going to create another table called weapons:

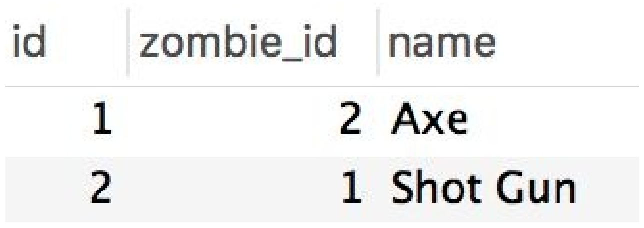
weapons

Field	Type	Length
id	INT	11
zombie_id	INT	11
name	VARCHAR	50

Notice the 'zombie_id' row in the table above.

This references the 'id' row in the Zombies table, and is referred to as a Foreign Key which occurs when a row in one table uniquely identifies a row in another table. This Foreign Key is what ties a relationship between the Weapons table and the Zombies table.

Let's say that we already have two weapons in our database that belong to each of our zombies:



id	zombie_id	name
1	2	Axe
2	1	Shot Gun

In the example above you can see that we have added two rows to our weapons table. We have added an "Axe" that belongs to a zombie with an ID of 2 and we have a "Shot Gun" which belongs to our zombie with an ID of 1.

Next we are going to display information about the Zombie, including their weapon; but first we need to create our Weapon Model, located at `app\Weapon.php`:

```
<?php namespace App;

use Illuminate\Database\Eloquent\Model;

class Weapon extends Model {

     protected $table = 'weapons';

}
```

Now, we could add the following to display all the information about our zombie including their weapon:

```php
<?php

use App\Zombie as Zombie;
use App\Weapon as Weapon;

Route::get('/zombie/{id}', function($id){
     $zombie = Zombie::find($id);
     echo 'Name: ' . $zombie->name . '<br />';
     echo 'Strength: ' . $zombie->strength . '<br />';
     echo 'Health: ' . $zombie->health . '<br />';

     $weapon = Weapon::where('zombie_id', '=', $zombie->id)->first();
     echo 'Weapon: ' . $weapon->name . '<br />';
});
```

We just introduced another new helper provided by the Eloquent library; this is the `where` function. Before we just used `find` which returned the object with an ID.

Above we are using `Weapon::where('zombie_id', '=', $zombie->id)- >first();`. What is happening here is that we want to get the weapon where the `zombie_id` is equal to our zombie id, and we want to get the first row.

You can learn more about all the different ways to retrieve data from our models by checking out the full documentation on Eloquent here: http://laravel.com/docs/eloquent.

The above example will get the job done; however, there is an even easier way of doing this. If we specify our relationship between the zombie and the weapon we can minify the amount of code we need to add. We could add our relationship to our Zombie Model, and that would look something like this:

```php
<?php namespace App;

use Illuminate\Database\Eloquent\Model;

class Zombie extends Model {

     protected $table = 'zombies';
     protected $fillable = ['name', 'strength', 'health'];

     public function weapon()
     {
          return $this->hasOne('App\Weapon');
     }
}
```

Above we are saying that a Zombie has one Weapon. We just create a new public function called `weapon()` and return the weapon.

So, with that addition to our Zombie Model we can now refactor our code and display all the information about our Zombie like this:

```php
<?php

```

```php
use App\Zombie as Zombie;

Route::get('/zombie/{id}', function($id){
     $zombie = Zombie::find($id);
     echo 'Name: ' . $zombie->name . '<br />';
     echo 'Strength: ' . $zombie->strength . '<br />';
     echo 'Health: ' . $zombie->health . '<br />';
     echo 'Weapon: ' . $zombie->weapon->name . '<br />';
});
```

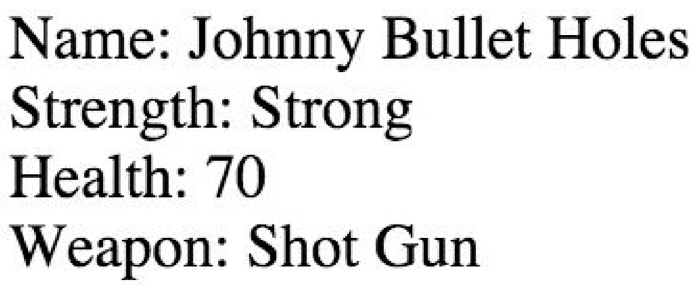
How great is that! By adding that relationship we just took these two lines of code:

```php
$weapon = Weapon::where('zombie_id', '=', $zombie->id)->first();
echo 'Weapon: ' . $weapon->name . '<br />';
```

and turned it into this one line:

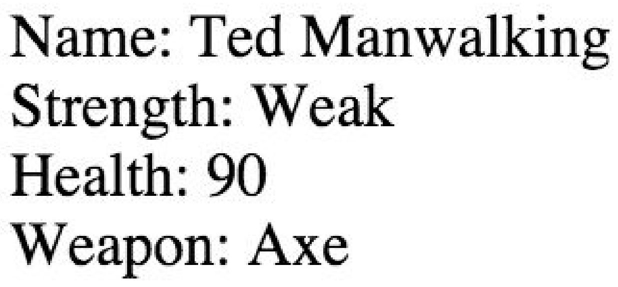
```php
echo 'Weapon: ' . $zombie->weapon->name . '<br />';
```

Being more readable and easier to work with, If we now visit (site.com/zombie/1) we should get our output which will look like the following:



```
Name: Johnny Bullet Holes
Strength: Strong
Health: 70
Weapon: Shot Gun
```

To see all the information about our zombie with an ID of 2 we could then visit (site.com/zombie/2)



```
Name: Ted Manwalking
Strength: Weak
Health: 90
Weapon: Axe
```

One last thing before we move on, what if we had the ID of our weapon and we wanted to see to which zombie it belonged? This can be achieved in the same way.

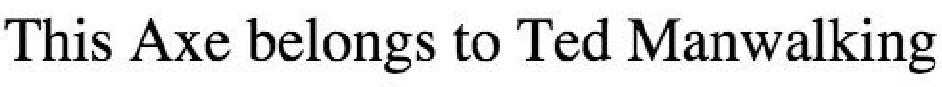
We can add our relationship in our Weapon class, but instead of the `hasOne` relationship, we will use the `belongsTo` relationship since a Weapon belongs to a zombie. To add this relationship we will add a zombie method to our Weapons model that will return a Zombie object like so:

```php
<?php namespace App;

use Illuminate\Database\Eloquent\Model;

class Weapon extends Model {

    protected $table = 'weapons';

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

Now, we could add the following route:

```php
use App\Weapon as Weapon;

Route::get('/weapon/{id}', function($id){
  $weapon = Weapon::find($id);
  echo "This " . $weapon->name . " belongs to " . $weapon->zombie->name;
});
```

We can now get to the zombie that owns this weapon by accessing `$weapon->zombie`. And if we were to visit (site.com/weapon/1) we would get the following output:



```
This Axe belongs to Ted Manwalking
```

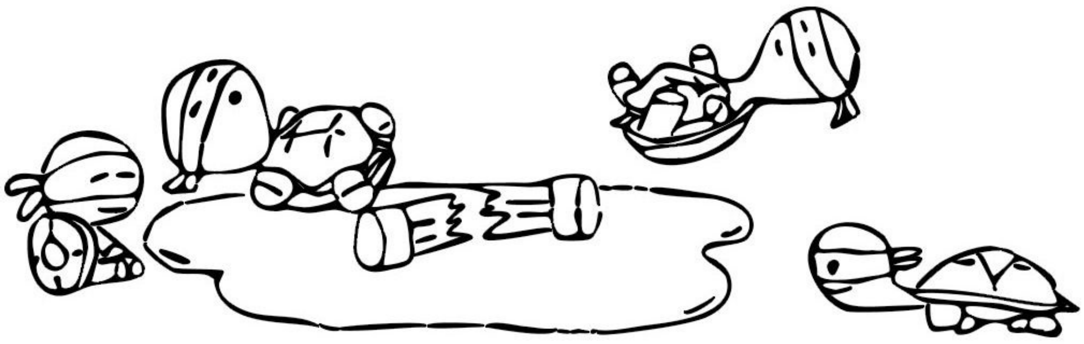
Using relationships in Laravel makes interacting with your data easy and fun.

The relationship between `hasOne` and `belongsTo` is referred to as a One-to-One relationship (since each of them has one of the other). If you wish to learn about the other relationships in depth head on over to http://laravel.com/docs/eloquent-relationships.

Models and relationships are a big part of what makes Laravel amazing. Instead of creating complex queries and bloated models, we can focus on more the fun stuff and build our application quicker than ever before.

Next up we are going to talk about mutators, which allow us to manipulate(mutate) data before it gets entered into or retrieved from the database.

Chapter 7 - Mutators



A zombie developer does not sanitize data when sending or receiving it from their database, whereas a Laravel developer uses mutators to manipulate data before and after getting data from the database.

Mutators are our friends, and they will help us defeat the zombie developer apocalypse by helping us sanitize or manipulate data in our Models.

Mutators

Mutators allow us to change data before sending it to or retrieving it from the database. When we manipulate data before it gets entered into the database, it is referred to as a *Mutator*, when we manipulate data after we retrieve it from the database it is referred to as an *Accessor*.

We are going to use a pretty simple example to show you the basics of using mutators.

We will use our zombies table as an example from the previous chapters. Let's say that anytime we get a zombie name from the database we want to make sure the name is capitalized. We could easily do this using an Accessor.

In our Zombie Model from our past examples we would want to add the following method:

```
public function getNameAttribute($value){
    return ucwords($value);
}
```

And now any time we get a zombie name from the database it will be capitalized. You will want to make the name of the function correspond to the name of the database row you want to change. Since our database row was `name` our function name is get**Name**Attribute.

This can be done just as easy using Mutators, except we will modify the data before it gets put into the database. To add this mutator we would add the following method to our Zombie model:

```
public function setNameAttribute($value){
    $this->attributes['name'] = ucwords($value);
}
```

The name would also be capitalized before we saved the data to the database. Our full zombie model with our Accessor and Mutator methods would look like the following:

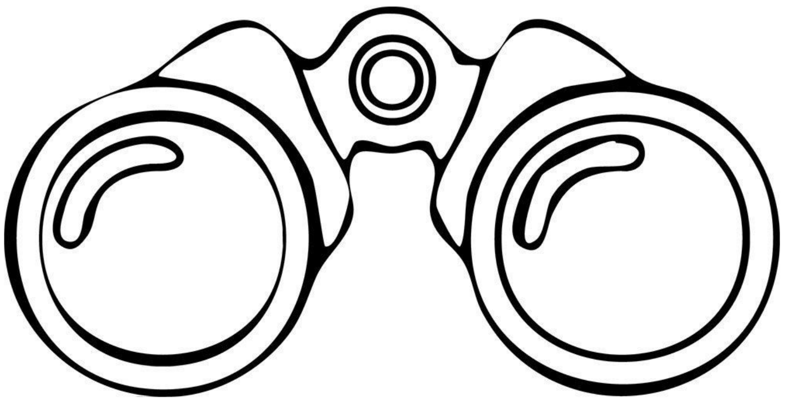
```
<?php namespace App;

use Illuminate\Database\Eloquent\Model;

class Zombie extends Model {

    protected $table = 'zombies';

    public function getNameAttribute($value){
        return ucfirst($value);
    }

    public function setNameAttribute($value){
        $this->attributes['name'] = ucwords($value);
    }
}
```

Most likely you wouldn't need the Accessor and Mutator to do the same thing, it would be a preference as to which one you wanted to use.

From the examples above, if anyone were to enter a zombie name into our database it would be capitalized. Hopefully, you can see the advantage of using mutators; it makes it easy to send and retrieve manipulated or formatted data from your database. Just another awesome functionality to help make our lives as developers easier.

Next up we are going to learn about views. Views are the files that typically contain the HTML & CSS. In the previous examples we have just been outputting our data from our `routes.php` file, but when we want to output data to the screen we will typically use our views. Let’s move on to the next chapter to learn more.

Chapter 8 - Views



A zombie developer entangles logic and HTML elements in the same page, whereas a Laravel developer separates their views from their logic.

Zombies are considered to be messy and not too friendly because they write code all over the place. As a Laravel developer we separate our data output from our data logic. In this chapter, we’re going to talk about using views. Leveraging the powerful features of Laravel will help us gain a clear VIEW of what we want to build (sorry for the cheesy pun).

Views

Views are essentially the HTML and the layout elements of a page. A view is what our user will see when they visit a page.

So, how might we go about creating a new VIEW for a specified route? Glad you asked, as it's very straightforward. Back in Chapter 2, we created a simple route called 'graveyard' that printed out a string, which looked like this:

```php
<?php

Route::get('graveyard', function(){
     echo 'Welcome to the graveyard!';
});
```

If we wanted to implement the functionality above with a view, our code would look like this:

```php
<?php

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

And this will load the file located at `resources/views/graveyard.php`. So, if we created that file and added the following code to this file:

```html
<html>
<head>
     <title>Welcome!</title>
</head>
<body>

     <p>Welcome to the graveyard!</p>

</body>
</html>
```

We would get the same functionality as the first view, except when using the view we would add valid HTML to our page, and this would be the correct way of structuring our code and displaying data on the screen.

All of your view files will be located inside of the `resources/views/`directory. Keeping our files well organized and in proper locations will help us keep our sanity and will make our jobs and lives much easier.

Now let's say that we wanted to get all the zombies from our database and pass them to our view, how might we do this. Simple enough, our route would look like this:

```php
<?php

use App\Zombie as Zombie;

Route::get('zombies', function(){
     $data = array('zombies' => Zombie::all());
     return view('zombies', $data);
});
```

And that would pass all our zombies to a view called zombies.php located in `resources/views/` folder. After creating a new file at `resources\views\zombies.php` we could list out our zombies by adding the following code:

```
<html>
<head>
    <title>Zombies</title>
</head>
<body>

    <ul>
        <?php foreach($zombies as $zombie): ?>
          <li><?php echo $zombie->name; ?></li>
      <?php endforeach; ?>
   </ul>

</body>
</html>
```

If we were to open up a web browser and navigate to site.com/zombies we would get an unordered list of our zombie names, which would look like the following:

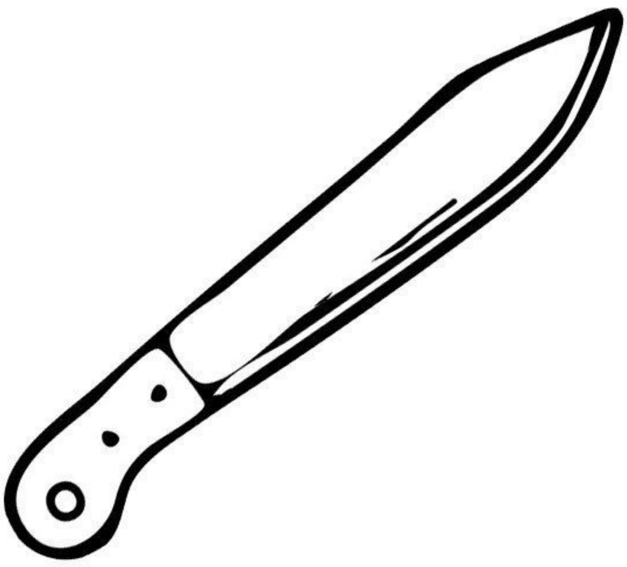
- Johnny Bullet Holes
- Ted Manwalking

To pass data to your views, you can just include all of the variables inside of an array that gets passed as a second argument in the view function.

```
return view('zombies', array( 'zombies' => Zombie::all() ));
```

Using views in Laravel is super easy, and it gets even easier if we choose to use the built-in blade templating engine. What is the blade templating engine? Let’s discuss that next.

Chapter 9 - Blade



A zombie developer’s code has long and combersome syntax, whereas a Laravel developer’s code contains clean and readable views by using the blade templating engine.

Using Blade Templating in your app will make your views look super clean and easily readable. Blade templating is optional to use, but after you get the hang of it, you’ll find yourself wanting to use it over and over again.

Blade

Blade templating is an easy way to render your views in a more readable fashion. Blade templating is a syntax that you can use in your view files that will then render into valid PHP code. Let me give you a quick example. Taking the `foreach` loop from our previous chapter we have the following code that displayed our zombies in an unordered list:

```
<ul>
     <?php foreach($zombies as $zombie): ?>
        <li><?php echo $zombie->name; ?></li>
    <?php endforeach; ?>
</ul>
```

If we wanted to use blade templating we could re-write that code using blade templating like so:

```
<ul>
     @foreach($zombies as $zombie)
          <li>{{ $zombie->name; }}</li>
     @endforeach
</ul>
```

And this makes it so much easier to read and write!

As you can see from the example above instead of doing a `foreach` and having to open and close your PHP tags, you can just use the `@` symbol. Additionally, when we add anything inside of `{{ }}` it will automatically be output to the screen.

The only thing that we need to do to use the blade templating engine is to rename our files from `.php` to `.blade.php`. So, from the example in the previous chapter we created a new file located at `resources\views\zombies.php`.

Instead, if we named it `resources\views\zombies.blade.php`, we could then use blade templating inside of that file.

Here is another quick example of using blade templating engine. Let's say that we have the following code in a blade.php view:

```
<?php if($var){ ?>
    The statement is true. variable = <?php echo $var ?>
<?php } else {
     The statement is false. variable = <?php echo $var ?>
<?php } ?>
```

All this does is print to the screen whether the statement is true or false, and then it displays the value of the variable. To clean this up using the blade templating engine, it could be written as follows:

```
@if($var)
     The statement is true. variable = {{ $var }}
@else
     The statement is false. variable = {{ $var }}
@endif
```

As you can see the above code is much cleaner and easier to read. All thanks to Blade Templates!

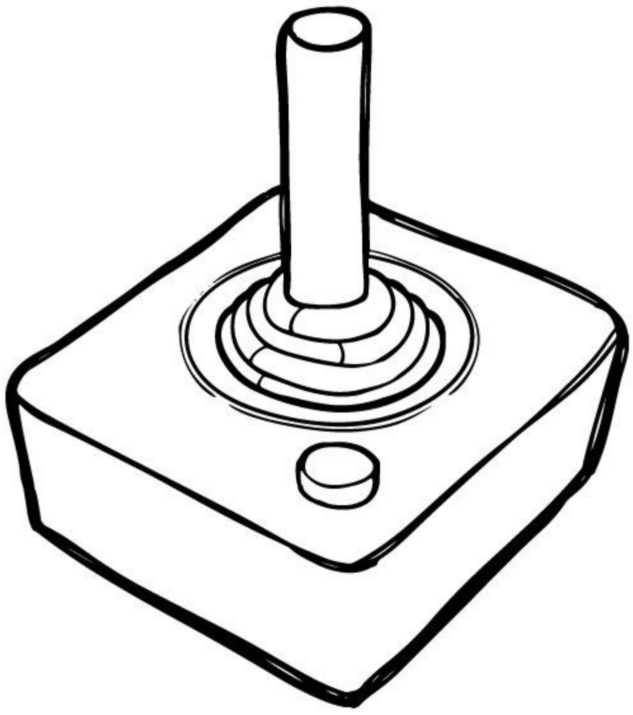
There are many other shorthands that you can use in your blade templates including layouts, loops, if statements, and much much more.

Be sure to checkout the full documentation on using all the blade shorthand syntax here: http://laravel.com/docs/blade

Now that we’ve got the hang of how Views work in a Laravel app we are going to move on to Controllers, which is where most of our functionality for our app will live.

Let’s read on and learn more about controllers.

Chapter 10 - Controllers



A zombie developer does not control situations well since their logic is all over the place, whereas a Laravel developer uses Controllers to separate their logic from the rest of their app.

Let’s get control of the situation and add some logic to our app.

Controllers can be looked at as the brains of the operation because this is where all the logic occurs.

Controllers

Models will get/set data from our database, the View is what's displayed in the browser, and the Controller is where the logic happens in our app. Your controllers for your application will live inside of the `app\Http\Controllers` folder.

Back in chapter 2, we had a section called 'Route Closures vs. Route Controller Actions' where a Route Closure creates an anonymous function, and you put the code directly in that function. This looks like the following:

```php
<?php

Route::get('/zombie', function(){
     echo 'Welcome to the Zombie Page!';
});
```

And a Route Controller, which maps the route to a controller, which looks like this:

```php
<?php

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

The example above will look to the `index()` method in the ZombieController.

So, let's go ahead and create a new file inside of `app\Http\Controllers` and call it `ZombieController.php` and insert the following code:

```php
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;

class ZombieController extends Controller
{
     public function index(){
          echo 'Welcome to the Zombie Page!';
     }
}
```

If we navigate to site.com/zombie we will get the same output of 'Welcome to the Zombie Page!'.

This is obviously a very simple example, usually the logic in your controllers will contain more code than what we did above.

> You may have noticed it seems like using a Route Closure was a much simpler way of doing things, but believe me, once your app starts to get bigger you will want to put all your logic inside of your Controllers and keep the routes.php file as clean as possible. Adding logic to your controllers keeps everything more organized and will make your workflow more efficient.

So let’s go ahead and explain the code in the ZombieController.php real quick. After opening our PHP tags we specify the namespace:

```
namespace App\Http\Controllers
```

The namespace is the current folder where our class is located, which is inside the AppHttpControllers folder.

Then, we are going to extend the Laravel default controller (You will typically use this same code in future controllers that you create). We need to tell our class which Controller class we need to use which is inside of the AppHttpControllers folder as well, so we will write:

```
use App\Http\Controllers
```

And we need to give our controller a name and say that we want it to extend from the default controller:

```
class ZombieController extends Controller
```

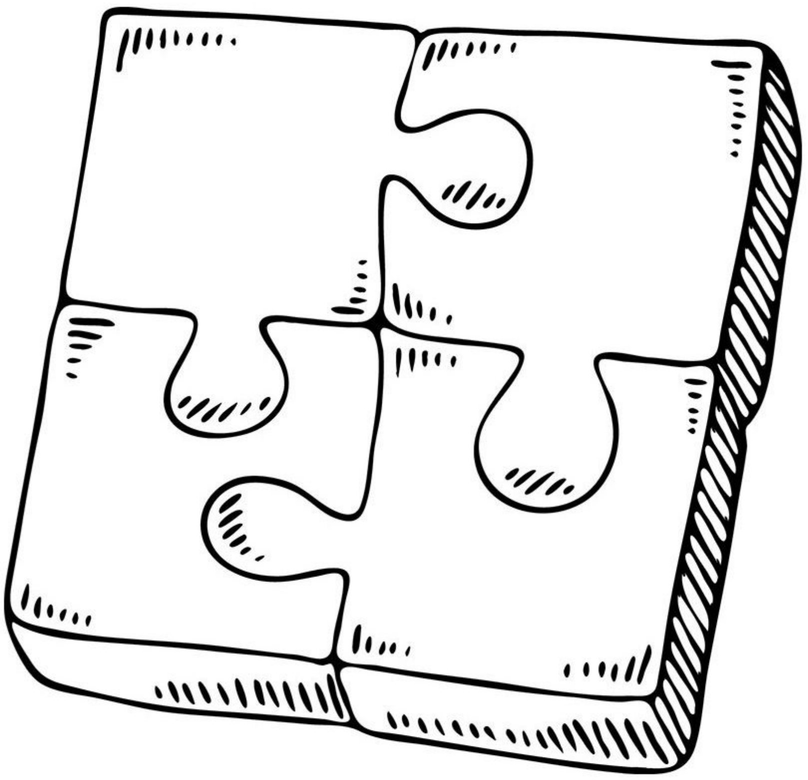
Finally, we need to specify our index() method and print out our message to the screen:

```
    public function index(){
        echo 'Welcome to the Zombie Page!';
    }
```

This is fantastic! We can map any route to any controller method. This will help keep our route file clean, and it will put the logic in our controller files.

In the next chapter, we will show you more about Controllers as we show you the process of piecing together functionality from the route to the controller and then to the view. Let’s continue and learn more about how these pieces come together.

Chapter 11 - Piecing It Together



A zombie developer may write all the logic and views in one file, whereas a Laravel developer uses routes, models, views, and controllers to piece their app together making it cleaner, more flexible, and better organized.

Leveraging all the different classes and functionality that Laravel offers will make our application more flexible. By separating our functionality in routes, models, views, and controllers, our code will be a lot cleaner and more enjoyable to work with.

Piecing It Together

In the last couple chapters, we talked about Routes, Models, Views, and Controllers. Now, let's go through an example of "Piecing those together" and show you how we might display a list of all our current zombies using routes, models, views, and controllers.

When we go to our `site.com/zombies` route, we want to be able to see all our current zombies. Here is the process that we will need in order to accomplish this. We will need to:

1. Create a route that maps to a controller method
2. Create our controller
3. Retrieve the zombies from the zombie model in our controller
4. Pass our zombie data to our view
5. Output our zombie data in our view

1. Create a route that maps to a controller method

This is fairly straightforward, we simply need to create a route inside of our `App\Http\routes.php` file that links to a controller method:

```php
Route::get('zombies', 'ZombieController@show');
```

2. Create a controller

We can use the same zombie controller that we created at

`App\Http\Controllers\ZombieController.php` which looks like this:

```php
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;

class ZombieController extends Controller
{
    public function show(){
        // Show our zombies
    }
}
```

Above you can see that instead of the `public function index()` we used `public function show()` because `show()` is the method that we used in the previous step.

3. Retrieve the zombies from the zombie model in our controller

Ok, inside of our Zombie Controller we need to get all our zombies from our Zombie model. We will add this code inside of the `show()` function and store it in a variable like so:

```php
    public function show(){
        // Show our zombies
        $zombies = Zombie::all();
    }
```

Great, we've stored all our zombies from our zombie model inside of the `$zombies` variable.

Before we go to the next step, I want to point out that our app will not know where to find `Zombie::all()`.

If you go back to chapter 5 where we created this zombie model we put it inside of the `App` namespace since it is in the `app` folder, so we would need to call `App\Zombie::all()`. Or we could just tell our controller which zombie to use, like so:

```
use App\Zombie as Zombie;
```

And now, Altogether our controller would look like this:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Zombie as Zombie;

class ZombieController extends Controller
{
    public function show(){
        // Show our zombies
        $zombies = Zombie::all();
    }
}
```

4. Pass our zombie data to our view

Passing our zombie data to our views is very straightforward. Inside of the `show()` function, we need to load a view file and pass along the zombie data. To load a view file we can return a view function at the end of our method like so:

```
return view('zombies', $data);
```

The first argument is a string with the view we want to load (located at `resources\views\zombies.blade.php`) and the second argument is an array of data that we wish to pass to our view.

So, all together we would want our `show()` function from our controller to look like the following:

```
public function show(){
    // Show our zombies
    $zombies = Zombie::all();

    // Store the zombies in an array of data
    $data = array('zombies' => $zombies);

    // Load the view and pass it our data array
    return view('zombies', $data);
}
```

Simple enough, now inside of our `resources\views\zombies.blade.php` we will have an array of zombies stored in a variable called `$zombies`.

> Notice that if we were to create a data array that looked like
>
> `$data = array('zombie guys' => $zombies)` we would then have an array of zombies available in a variable called `$zombie_guys`.

The last thing we need to do is to display our zombies in our views.

5. Output our zombie data in our view

Inside of our zombie view file located at `resources\views\zombies.blade.php` we can add a basic HTML page with an unordered list and loop through each of our zombies like so:

```
<html>
<head>
    <title>Zombies</title>
</head>
<body>

    <ul>
        @foreach($zombies as $zombie)
            <li>{{ $zombie->name; }}</li>
        @endforeach
    </ul>

</body>
</html>
```

In the example above we do a simple foreach statement and loop through each of the zombies and list out their name.

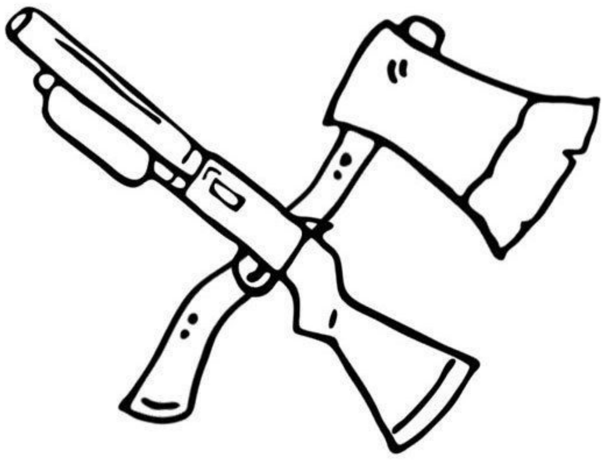
Also, notice that above we are using blade syntax that we covered previously.

There you go! That was a fundamental overview of how each piece will work from your route, model, controller, and view.

Pretty fun stuff, right!

Now that we have a basic overview of how our app works let’s move on to talking about an excellent helper tool in Laravel called `artisan` that will make our lives much easier.

Chapter 12 - Artisan



A zombie developer creates files manually, whereas a Laravel developer uses artisan to help generate files and functionality for them.

When fighting a zombie would you rather have a shotgun or an axe?

Probably a shotgun, it would make killing zombies much faster and a lot easier. Well, if there were an easier weapon or tool to use that would make creating your app faster and easier wouldn't you want to use it?

That's exactly what Laravel's `artisan` command provides for us.

Artisan

We've used the `artisan` command in previous chapters a handful of times, so you might have a basic idea of what this command does.

A quick and short definition of `artisan` is that it is a command line tool that can help generate files and run php commands.

If you run `php artisan` in a command prompt, you will see a list of available commands available to use.

In this chapter, we are only going to go over a few basic commands to get you started. First let's talk about how you can create new files.

Making new files with Artisan

Using the `php artisan make` command we can generate files that we need for our app.

In the last chapter and back in chapter 5 when we needed a new Controller we just created a new file and started adding our code. Well, instead of manually creating the file we could have run an artisan command to do this for us:

```
$ php artisan make:controller ZombieController
```

And if we look inside of our `App\Http\Controllers` folder we will see a new file called `ZombieController.php` that has the following code inside of it:

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

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

class ZombieController extends Controller
{
    //
}
```

How great is that! We just ran one command, and our controller is ready for us to start adding methods!

We could also have used a similar command to create our Zombie Model, like so:

```
$ php artisan make:model Zombie
```

And now if we look inside our `app` folder we will see a new file called Zombie.php with the following contents:

```php
<?php

namespace App;

```

```
use Illuminate\Database\Eloquent\Model;

class Zombie extends Model
{
    //
}
```

And just like that we’ve created our model class that can interact with our `zombies` table in our database.

> Quick side note: if you want this Model to interact with another database table you could always add a protected variable called table, like so `protected $ table = 'zombie_folks';` and now this model will interact with the ‘zombie_folks’ table in your database.

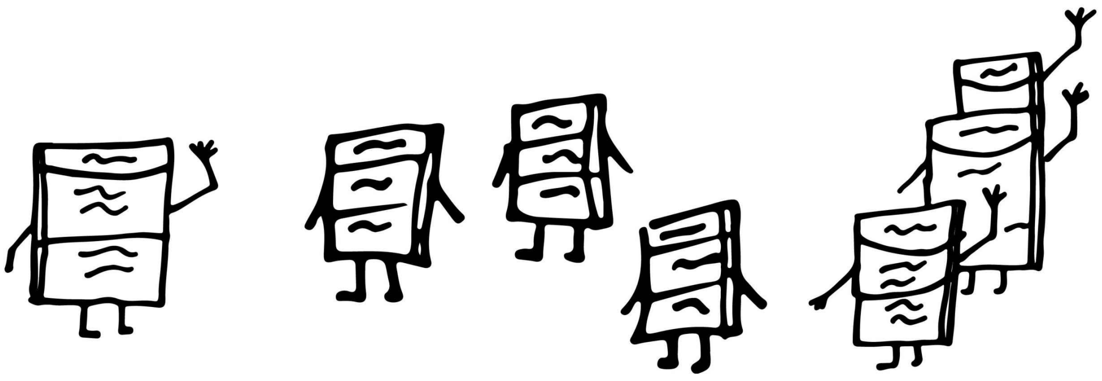
It’s amazing how much easier our Laravel programming will be if we leverage the power of `artisan`.

In this chapter we only covered 2 basic uses of the artisan command `php artisan make:controller` & `php artisan make:model`, but there are so many more artisan commands for you to master and add arsenal of helpers as you program the next greatest app.

Be sure to head on over to https://laravel.com/docs/artisan to learn all about artisan and all the other commands you can use.

In the next chapter, we are going to cover middleware, which is a great way to run code between page requests.

Chapter 13 - Middleware



A zombie developer does not have any functionality between page requests, whereas a Laravel developer leverages Middleware to keep their app more secure and flexible.

You can think of middleware as the gatekeeper for requests. Before our app allows a request it must first pass through the gatekeeper. If the gatekeeper grants the user access the app will move on to the next request; however, if the user is denied access then the gatekeeper will not allow them to pass.

Middleware

Middleware is an easy way to run functionality betwee HTTP requests. An example middleware might disallow users access to certain routes if they are not authenticated.

As an example let's run some functionality to check if our current user is a zombie or not. If the user is a zombie, they are not allowed to access the route. For the sake of simplicity we are going to hard-code a variable to be true(1) or false(0) if the user is a zombie. Check out the following route:

```
Route::get('arsenal', function(){
     $is_zombie = rand(0, 1);

     if($is_zombie){
          return redirect('/home');
     }

     // If the user is not a zombie we can run any code below
});
```

So, in the code above we are saying that if the user is a zombie we want to redirect them to the homepage route. Now, if we wanted another route we could do the same thing here:

```
Route::get('armory', function(){
     $is_zombie = rand(0, 1);

     if($is_zombie){
          return redirect('/home');
     }

     // If the user is not a zombie we can run any code below
});
```

And now, any user who is a zombie will not be allowed access to any of those routes. Let's say we wanted to create one more route… Uhhh… We have to do the same code again…

Instead of continually adding the same code to protect against a zombie user we could instead create a **Middleware** to handle this functionality for us. We can use our good friend artisan to create a new middleware file for us. Run the following in your command line:

```
$ php artisan make:middleware isNotZombie
```

And now you should see a new file inside of `app\Http\Middleware\isNotZombie.php`, which has the following contents:

```
<?php

namespace App\Http\Middleware;

use Closure;

class isNotZombie
```

```
{
    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        return $next($request);
    }
}
```

In the code above you can see that we have a function called `handle`, and inside this function is where we will add our functionality to check if the user is a zombie or not, so we'll go ahead and change the `handle` function above to look like the following:

```
public function handle($request, Closure $next)
{
    $is_zombie = rand(0, 1);

    if($is_zombie){
        return redirect('/home');
    }

    // If the user is not a zombie we can run any code below
    return $next($request);
}
```

Above we are running the same functionality to make sure the user was not a zombie. In the code above if the user is a zombie they will be redirected to the homepage; however, if not, we just continue to the rest of the code.

The final step is to register our middleware in our app by giving it a specific name and adding our middleware class to our application Kernel located in `app\Http\Kernel.php`.

We will need to add our class to the `protected $routeMiddleware = []` array like so

```
protected $routeMiddleware = [
    'auth' => \App\Http\Middleware\Authenticate::class,
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::c\
lass,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
    'zombie' => \App\Http\Middleware\isNotZombie::class,
];
```

Notice above there are already a few middleware routes that come packaged with Laravel. In fact, we'll be learning more about the 'auth' middleware in the next chapter.

Now, to run our middleware we can create a Middleware Group and add our routes inside:

```
Route::group(['middleware' => ['zombie']], function () {
     Route::get('arsenal', function(){
          // If the user is not a zombie we can run any code below
     });
```

```
    Route::get('armory', function(){
        // If the user is not a zombie we can run any code below
    });
});
```

This is great, now any routes that we want to protect against any user that is a zombie can be wrapped in the zombie middleware group.

Web Middleware

If you looked inside the `routes.php` of a new Laravel app, you may have noticed that you already had a middleware group available to you, which was the `web` middleware:

```
Route::group(['middleware' => ['web']], function () {
    //
});
```

This middleware will allow you to have access to sessions and certain kinds of security in your app. All routes in your web app will typically be placed inside of this middleware group.

Multiple Middleware

We can even nest groups inside of each other. The following will group our routes in the `web` and the `zombie` middleware:

```
Route::group(['middleware' => ['web']], function () {
    //
    Route::group(['middleware' => ['zombie']], function () {
        // Add our routes here
    });
});
```

Fantastic! But we could make this even more efficient if we want to include both middlewares in the same group:

```
Route::group(['middleware' => ['web', 'zombie']], function () {
    // Add our routes here
});
```

You can see that we can add an array of middleware classes in our group, and they will all be run before we access the specified routes within.

Route Specific Middleware

As an alternative to using Middleware Groups we can add Middleware to each route specifically, like so:

```
Route::get('arsenal', ['middleware' => 'zombie', function () {
    //
}]);
```

We could also chain the middleware method to our route definition, like so:

```
Route::get('arsenal', function () {
    //
})->middleware(['first', 'second']);
```

You can see there are multiple ways to add your middleware and it depends on the way that works best for you.

Controller Middleware

You could also specify Middleware in a controller.

The methods in that controller will always run the Middleware before running any method. Let’s say that our routes above linked to a controller method, like so:

```
Route::get('arsenal', 'WeaponsController@arsenal');
Route::get('armory', 'WeaponsController@armory');
```

Our above routes will use methods from a WeaponsController. So let’s create that ‘WeaponsController’ by running the following artisan command:

```
$ php artisan make:controller WeaponsController
```

And a new file will be created at `app\Http\Controllers\WeaponsController.php` with the following contents:

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

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

class WeaponsController extends Controller
{
    //
}
```

Inside of our new class we can create a constructor, and easily specify any type of middleware like so:

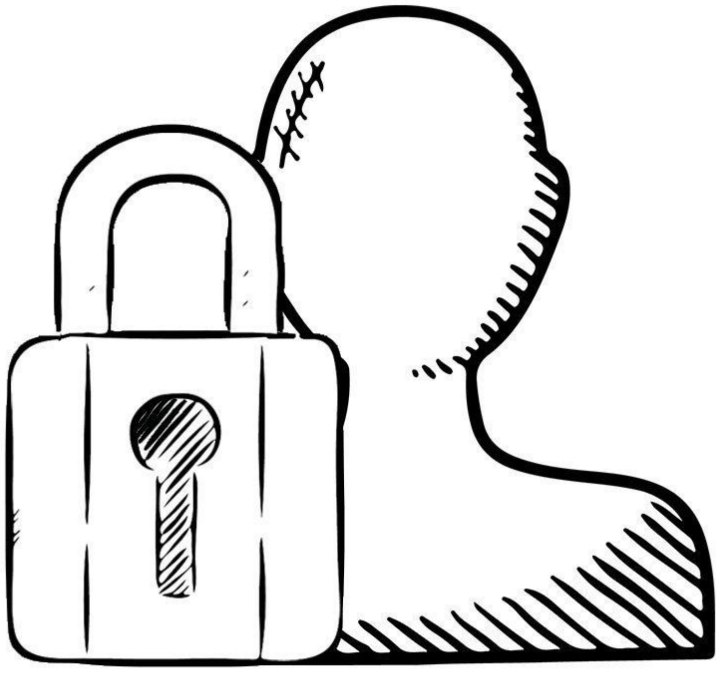
```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

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

class WeaponsController extends Controller
{
    //
    public function __construct()
    {
        $this->middleware('zombie');
    }
}
```

Any WeaponsController method will run through this Middleware before being executed. This kind of flexibility makes it easy to add functionality between any HTTP request of any controller.

We could always get by without using middleware in our projects, but after you get the hang of how to use them it will make your code more flexible, readable, and fun to work with.

Let’s move on now to Authentication and you’ll see another example of middleware and how we can protect certain routes from only being accessible to authenticated users.

Chapter 14 - Authentication



A zombie developer spends a few weeks building authentication into their apps, whereas a Laravel developer uses the built-in feature to add authentication to their app in minutes.

During the zombie developer apocalypse, you will need to be very cautious. Before letting anyone inside your home you probably need to authenticate that they are not a zombie. Similarly, when building your app you will most likely need to verify certain users before allowing them access to certain pages.

This is called authentication and it allows or disallows certain types of users to different sections of your app.

Authentication

When you login to an account on any website, it is referred to as authentication.

When you build your application in Laravel you will possibly need a way to allow your users to login to your application. This feature is built-in with Laravel and can be created in a matter of minutes.

In this chapter we'll be going over just how easy it is to add authentication to your Laravel app. Let's start out with a new laravel application:

```
$ laravel new my-first-authentication
```

Then you will need to (change directory) into your new laravel app:

```
$ cd my-first-authentication
```

Next, we'll run an artisan command to setup our authentication files.

```
$ php artisan make:auth
```

That command will generate a few views, controllers, and routes. If everything went as planned with that command you will see a message that says `Authentication scaffolding generated successfully!`.

Now, we only have a couple more steps to finish our full authentication system in our app. We need to connect a database to our application and add our users table to the database.

To connect a database to your laravel app you will need to modify a file located in the root of your directory. This is called the `.env` file.

> If you don't see this file, then hidden files may not be visible on your machine. Any file that starts with a . is typically hidden on most machines. If you google `How to show hidden files on Mac/Windows`, you'll find a quick solution that will show hidden files :)

So, open up the `.env` file and you will see something that looks like the following:

```
APP_ENV=local
APP_DEBUG=true
APP_KEY=ggMXvaUVlsxpMh8Jq8EM2icpL7isyWzl

DB_HOST=localhost
DB_DATABASE=homestead
DB_USERNAME=homestead
DB_PASSWORD=secret

CACHE_DRIVER=file
SESSION_DRIVER=file
QUEUE_DRIVER=sync

REDIS_HOST=localhost
REDIS_PASSWORD=null
REDIS_PORT=6379
```

```

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

The only part we will be concerned with is the DB_HOST, DB_DATABASE, DB_USERNAME, and DB_PASSWORD. For each of these variables, we will need to add in our local database credentials.

If you haven't already, you will need to create a new database. Let's say that we called this database `my-first-authentication`. We would then need to update our credentials like so:

```
DB_HOST=localhost
DB_DATABASE=my-first-authentication
DB_USERNAME=root
DB_PASSWORD=root
```

And above your database username and password will depend on your environment.

Alright, so that's it as far as connecting our database to our application.

The final step is going to be to add the users table to our database. We are going to do this by using our `artisan` helper and another new concept called migrations we'll go into further details on `migrations` in chapter 17.

Let's run the following command in our command prompt:

```
$ php artisan migrate
```

If all went well you'll see an output similar to:

```
Migration table created successfully.
Migrated: 2014_10_12_000000_create_users_table
Migrated: 2014_10_12_100000_create_password_resets_table
```

And if you checkout your `my-first-authentication` database you'll see that you now have a couple of tables.

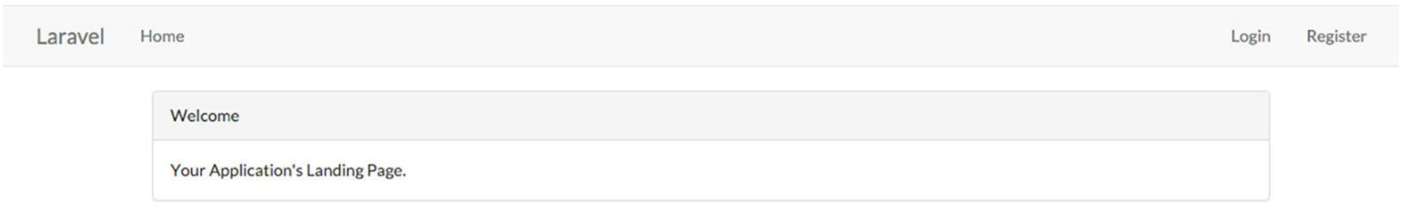
One last step. Make sure that all your current routes are wrapped inside of the 'web' middleware group we talked about in the previous chapter.

Your routes.php file should look like this:

```php
<?php

Route::group(['middleware' => 'web'], function () {
    Route::get('/', function () {
        return view('welcome');
    });

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

AND BOOM! THATS IT!

We’ve just added a full authentication system in our laravel up. If you run your laravel app in a browser you’ll see that you now have a new homepage:

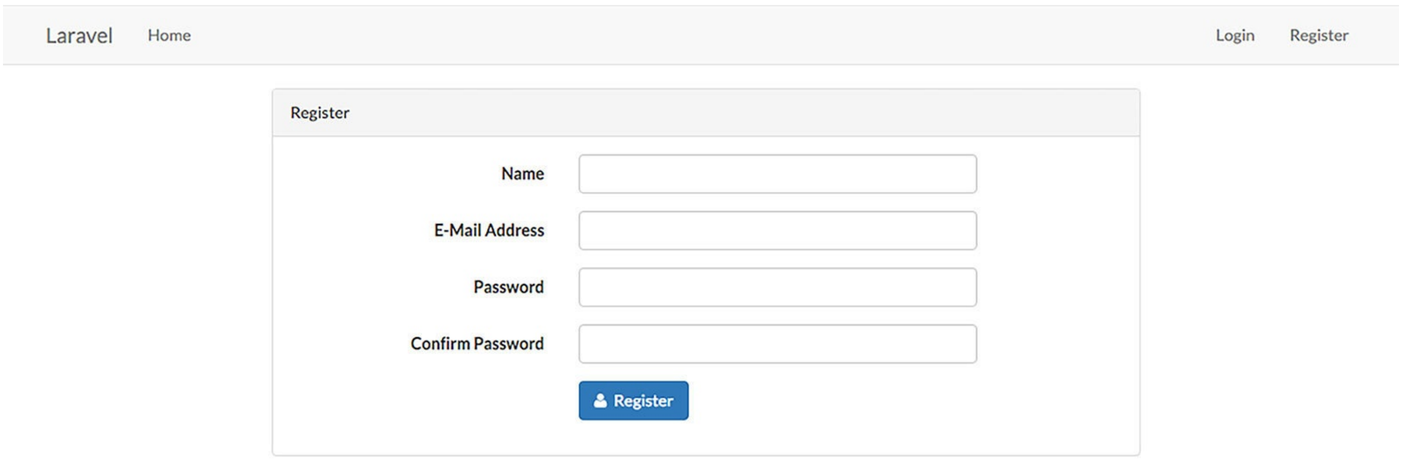


Welcome

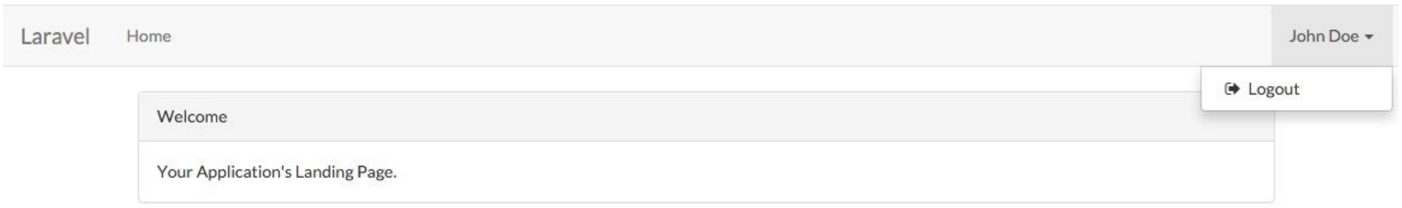
Your Application's Landing Page.

> Quick Note: If you don’t have a virtual host (local URL) setup with your laravel app you can always use the artisan tool to start up a quick server. In the root of your application, you can run `php artisan serve` and your app will be available in a browser if you navigate to `http://localhost:8000`

On your new Laravel homepage you’ll have a few buttons on your navigation. You can click on the register button and you will have a registration form in front of you:



Go ahead and register for an account and the app should automatically log you in and redirect you to the homepage. As you can see you now have a dropdown at the top right of your app that contains your username. If you click on the dropdown you will see a logout button:

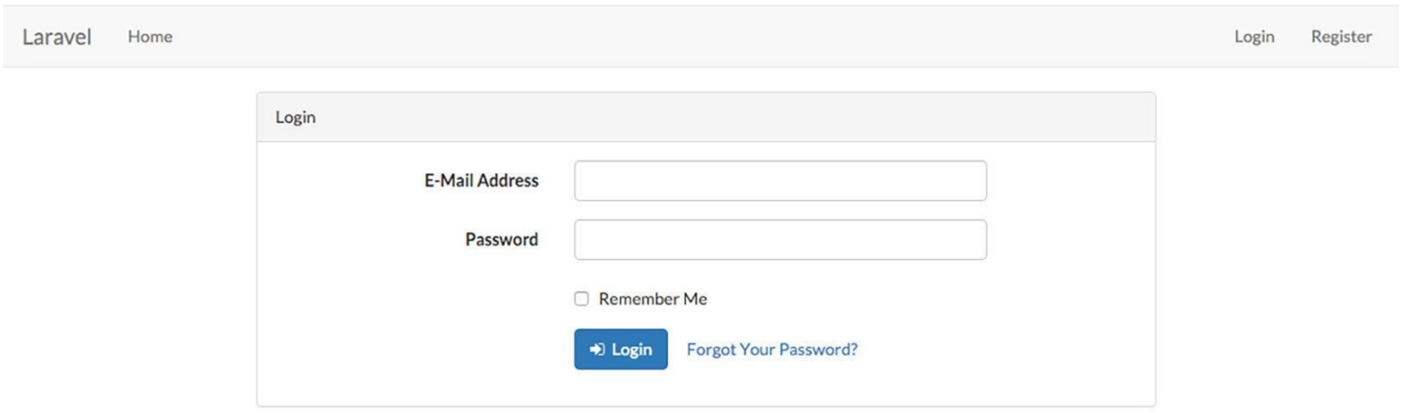


Welcome

Your Application's Landing Page.

Before you logout let’s go ahead and visit a route at: `site.com/home`. It will give you a message that you are currently logged in. Try going up to your username dropdown in the right hand corner and click the logout button. You will then be redirected back to the main URL. If you try visiting the `/home` URL you will no longer be able to access it.

Additionally, to access the login page you can always click on the login button on the top right.



How great is that? Our full authentication system is already built for us!

Let's do one last thing to show you how to check if a user is authenticated. Let's create a route that will show the current authenticated user profile.

To do this we would create a new route in the 'web' middleware group inside of `app\Http\routes.php`, which looks like this:

```
Route::get('profile', 'UserController@profile');
```

So, our 'profile' route links to a UserController class that we will need to create at `app\Http\Controllers\UserController.php` and that would look like this:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

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

class UserController extends Controller
{
    public function profile(){
        echo 'welcome to your profile';
    }
}
```

Now, if we visit that route `/profile` we will see a message that says 'welcome to your profile'.

Great so far.

Inside of this function let's check to see if the user is logged in and display their email address; otherwise, we will redirect them to the homepage.

```php
public function profile(){
    if (Auth::check()) {
        echo 'Welcome to your profile<br />';
        echo Auth::user()->email;
        echo '<br /><a href="/logout">Logout</a>';
    } else {
        return back();
    }
}
```

In the above code we can use the 'Auth' class and all the helpers it provides. So we can use the following code to check if the user is logged in or not.

```php
Auth::check()
```

The `Auth::check()` method will return true if the user is logged in or it will return false if the user is not logged in.

Another helper method that we used in the code above is the `Auth::user()` method and this will return the current logged in user object. We could easily echo out the users email address by doing this:

```php
$user = Auth::user();
echo $user->email;
```

Or all in one line we could write it like this:

```php
echo Auth::user()->email;
```

Make sure to note that we are using the Auth class above. We need to specify that we want to use the Auth class in this file by adding this to the top of the UserController file:

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

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

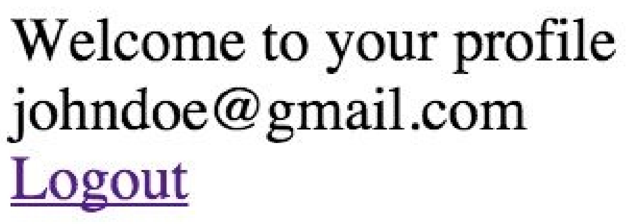
Make sure that you only run the `Auth::user()` method after you have checked that the user is logged in.

From the example above, if the user is not logged in we want to redirect to them back to their previous URL:

```php
return back();
```

The `back()` call is a helper function that will redirect the user back to the previous page.

Moving on, if we were to visit the '/profile' route in our app and we're logged in we would see a simple output like the following:


Welcome to your profile
johndoe@gmail.com
Logout

Otherwise, if we are not logged in we will be redirected back to our previous URL.

This code works just fine, but in most cases, we want to make sure that we aren't outputting data from our controller, that's what views are for.

Let's rewrite our UserController.php to look like the following:

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

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

class UserController extends Controller
{
     public function profile(){
          if (Auth::check()) {
               $user = Auth::user();
               $data = array('user' => $user);
               return view('profile', $data);
          } else {
               return back();
          }
     }
}
```

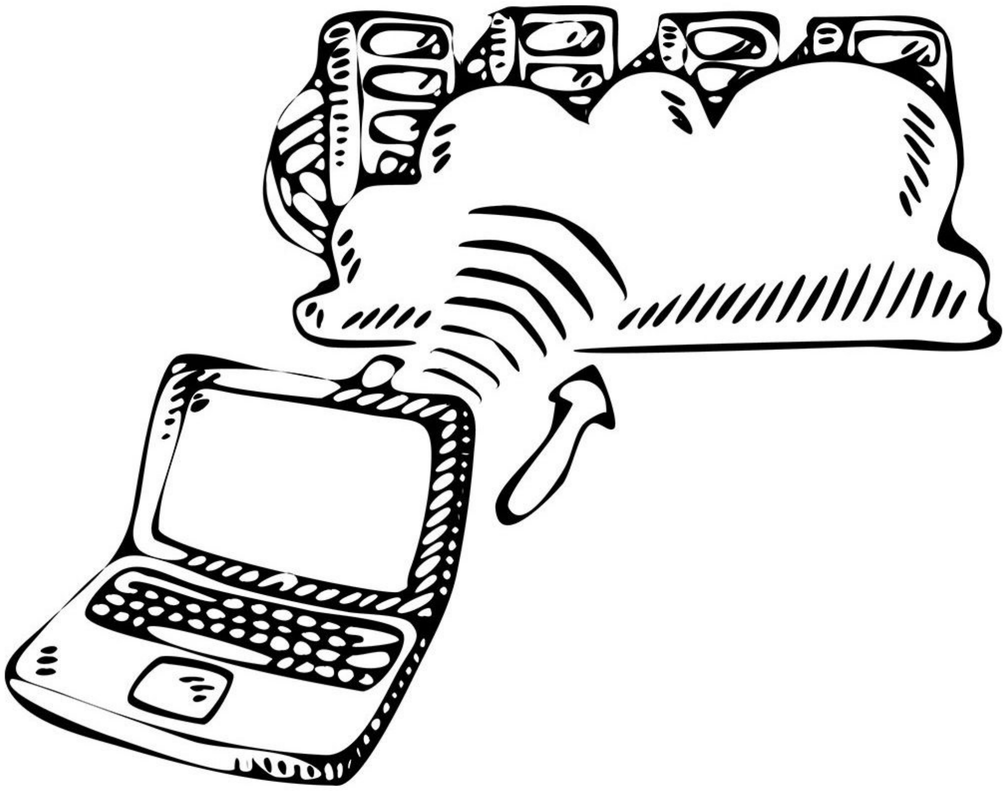
Inside of our `profile()` function, if the user is authenticated we get the user object and pass it in an array of data to a view called `profile`. So, now we need to create a new view file located at `resources\views\profile.blade.php` with the following contents:

```html
<!DOCTYPE html>
<html>
<head>
     <title>Your Profile</title>
</head>
<body>
     <p>Welcome to your profile</p>
     <p>{{ Auth::user()->email; }}</p>
     <a href="/logout">Logout</a>
</body>
</html>
```

And the code above will print out a valid HTML page with the similar output from the screenshot above.

I hope you can see the power of this built-in authentication system and how much time it will save you in the future. If your app needs an authentication system, you can have this fully integrated within minutes thanks to Laravel!

Chapter 15 - Requests



A zombie developer doesn't handle requests efficiently, whereas a Laravel developer uses the built-in Request class to handle input sanitization and security.

Requests allow our app to receive messages from the client. If these are not handled well or efficiently it can lead to a deteriorating user experience and could have significant vulnerabilities.

Requests

Laravel is built on top of PHP, which is a server-side scripting language.

This server side language can accept requests from the client (web browser) such as submitting a form or even requesting a route URL.

When a user types in a URL in their browser, the browser will then contact or send a request to the server. So, when data is sent from the browser to the server it is called a 'Request'.

We used the request object in previous chapters. Back in chapter 4, we used it to capture the id of a zombie:

```php
<?php

use Illuminate\Http\Request;

Route::delete('/zombie', function(Request $request){
     $id = $request->input('id');
     Zombie::destroy($id);
});
```

So, whenever we want to use the Request Class we need to make sure to specify that we want to use the `Illuminate\Http\Request` namespace, then we can simply pass the (Request $request) object as a parameter.

The request class offers us a ton of awesome information including:

- The Request URI
- The Request Method
- The Request Input
- The Request Cookies
- The Request Files

The Request URI

To get the current request URI we could do the following:

```php
$uri = $request->path();
```

So, if we tried to access the following route: `site.com/zombie/1`, the `$uri` in the above example would be `zombie/1`. Another cool thing that we can do is check if we are currently on a particular route, like this:

```php
if ($request->is('zombie/*')) {
    // we have hit the zombie/{id} route
}
```

Inside of the `is` method we can use an asterisk `*` as a wildcard.

If we wanted to fetch the full URL instead we could simply get it like this:

```
$url = $request->url();
```

Request Method

Now, let’s say we wanted to figure out what kind of request this is.

Is it a GET, POST, PUT, or DELETE method? Easy peasy, we can do that like this:

```
$method = $request->method();
```

Or we could use the isMethod function to check if it is a certain request:

```
if ($request->isMethod('post')) {
    // we have a post method
}
```

The Request Input

This is the request method that we have used in the previous chapters. This is where we get input that has been submitted via a form like this:

```
$zombie_name = $request->input('name');
```

We could also get all the input data as an array by doing the following:

```
$input = $request->all();
```

Finally, we could do a quick check to see if the request has a certain input value:

```
if ($request->has('name')) {
    //
}
```

The Request Cookies

To retrieve a cookie value from our request we could simply do the following:

```
$cookie = $request->cookie('name');
```

The Request Files

We could also use the Request class to retrieve uploaded files like so:

```
$file = $request->file('name');
```

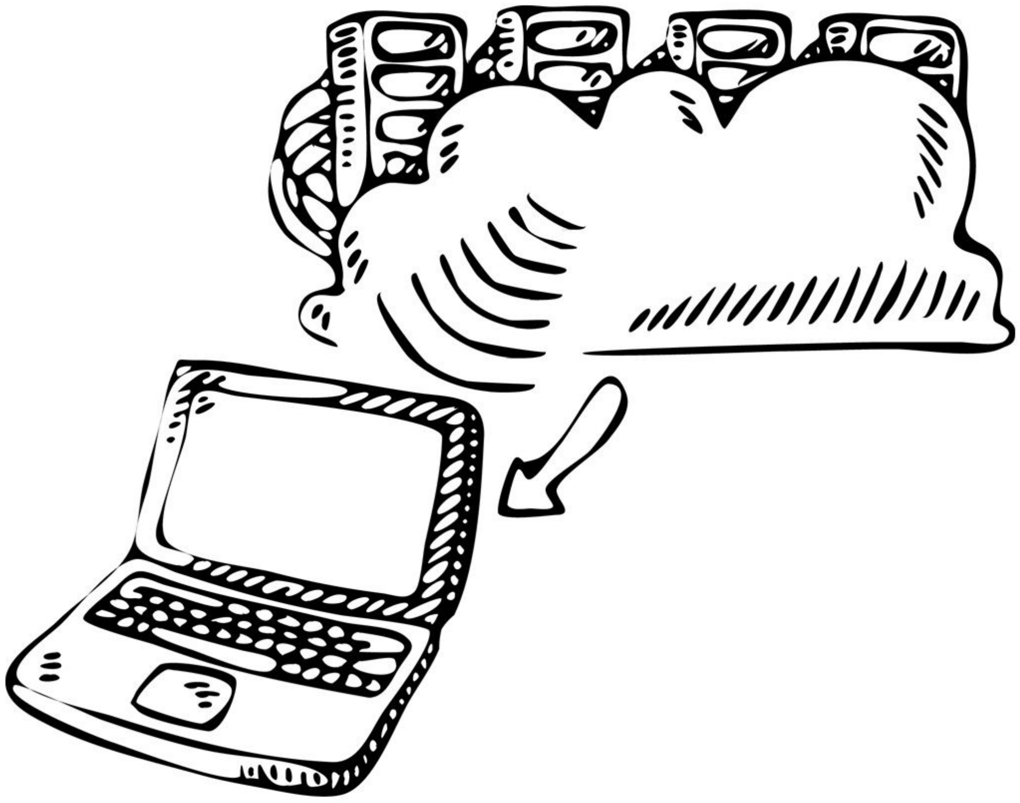
And we could check if the request has a file:

```
if ($request->hasFile('name')) {
    //
}
```

There are a few more request types that are provided by the Laravel Request class. Be sure to checkout the full documentation (https://laravel.com/docs/requests) to learn more.

So, we just learned all the awesome things that the Request class offers and we know how our app can accept requests, but what can it do after it gets a request? Well, it can also send a response. Our application can receive input, and it can also send output. Let’s learn more about the output that our application can send.

Chapter 16 - Responses



A zombie developer does not respond well to their clients, whereas a Laravel developer sends clear and concise responses.

When we refer to **Responses** we are primarily referring to the output that we send to the client. When our Laravel app receives a request, it can then send a response.

Responses

In the previous chapter, we talked about requests and how our Laravel app can accept them. In this chapter we are going to go over the Responses our app can send.

When our app get's a request, it can also return a response.

Here's a simple example of a response:

```
Route::get('/apocalypse', function () {
    return 'End of the World!';
});
```

In the example above our laravel app retrieves the *Request* of a GET method to the `apocolypse` route and it simply returns a string as the *Response*. This is the simplest form of a response.

So, when we return a response that contains HTML our Laravel app is essentially returning an HTML document that gets displayed in the users browser.

Attach a Cookie to a Response

In the previous chapter we talked about how your app can retrieve a cookie, now lets see how we can set a cookie by attaching it to a response. Remember a cookie is stored on the client side (browser), so when our app sends a response it will need to attach a cookie to be set by the browser.

We can easily do this by attaching it to the end of a view response like so:

```
Route::get('set_cookie', function(){
    return response()->view('apocalypse')->withCookie('name', 'value');
});
```

JSON responses

If we wanted to output a JSON response for a particular route we could use the following syntax:

```
Route::get('json_response', function(){
    $data = array(
        'name' => 'Johnny Bullet Holes',
        'strength' => 'strong');

    return response()->json( $data );
});
```

By running the route above, we get a nice JSON output as the response.

Lastly, we can perform a redirect as a response like so:

```
Route:get('redirect_me', function(){
    return redirect('zombie/1');
});
```

The route above will redirect to the `site.com/zombie/1` route as a response.

It’s pretty straightforward, right? Our application gets requests and sends responses. There are a few more responses that you may want learn about by checking out the docs here https://laravel.com/docs/responses.

Next, let’s move on to something exciting called migrations.

Migrations are a way of storing our database schemas in files so they can easily be versioned, shared, and backed up.

Chapter 17 - Migrations

A zombie developer exports and imports SQL files, whereas a Laravel developer uses migrations to import and export database schemas.

Exporting and importing SQL files are dead, anyone who performs these actions have zombie like tendencies.

There is a new way of doing data backup which is called Migrations.

Migrations

If you have ever backed up any of your PHP web apps, you are probably familiar with performing a MySQL dump backup, which is a dump of all your database data into a single file. This way when you want to restore your app you have to import all the database data.

If you are familiar with working on a team and someone adds a new row to the database, they'll have to send you over an updated database schema which may conflict with another modification.

Bleh… It just gets too complicated managing multiple iterations of your database.

Well, thanks to migrations, MySQL dumps will be a thing of the past and sharing your updated schemas with teammates will be super simple. By using migrations, you can create files in your app that store the database structure.

If you remember back in chapter 5 we created a zombies table that looked like the following:

zombies

Field	Type	Length
id	INT	11
name	VARCHAR	50
strength	VARCHAR	20
health	INT	3
created_at	TIMESTAMP	
updated_at	TIMESTAMP	

Usually to create a database table we would open up phpMyAdmin, Sequel Pro, or some other kind of sequel application and add each of these rows one at a time.

Alternatively if we are using migrations we can leverage our good 'ol friend `artisan` to help us with this.

Let's create our first migration using the artisan tool:

```
$ php artisan make:migration create_zombies_table
```

After running the following command, you will see a new file that has been created inside your project located in the `database\migrations` folder, and it probably looks something like:

`XXXX_XX_XX_XXXXXX_create_zombie_table.php`, the `X's` resemble a datetimestamp.

Make sure to notice that there are already 2 files in this folder which are `*_create_users_table.php` and `*_create_password_resets_table.php`. These are migrations that come prepackaged with Laravel and are used to create the tables used by the built-in authentication functionality we discussed in Chapter 14.

Let’s open up our new zombie migration file and you should see something similar to the following:

```php
<?php

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

class CreateZombieTable extends Migration
{
     /**
     * Run the migrations.
     *
     * @return void
     */
     public function up()
     {
          //
     }

     /**
     * Reverse the migrations.
     *
     * @return void
     */
     public function down()
     {
          //
     }
}
```

Notice that there are 2 methods available in our new migration class.

We have an `up()` method and a `down()` method. The `up` method is used to modify something in our database and the `down` method is used to reverse what we did in the `up` method.

Let’s add some code to the `up()` method that looks like the following:

```php
public function up()
{
     Schema::create('zombies', function (Blueprint $table) {
          $table->increments('id');
          $table->string('name');
          $table->string('strength');
          $table->tinyInteger('health');
          $table->timestamps();
     });
}
```

Now, if we have an empty database and we were to run the following command:

```
$ php artisan migrate
```

All our migrations will then be run and we should see a few tables inside our database including the `zombies` table.

If we were to add the following to our down function:

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

And we run:

```
$ php artisan migrate:rollback
```

It will run our `down()` method and we will no longer see our zombies table in our database.

Hopefully, you can see the power of migrations. We could easily share a GitHub repo with another user and they could pull our code down and create a new database. Then they can run the migrations and they will have the most up-to-date database schema.

There are many other types of functionality you can run in your `up` and `down` methods, be sure to checkout the full documentation on migrations at Laravel's documentation to learn more.

Fantastic!

Migrations allow us to save and version our database schema in our project files! What about the actual data that gets inserted into our database? What if we had some data that we wanted to `seed` into our database schema. Simple enough we can do that by using Seeds.

Chapter 18 - Seeds

A zombie developer imports data into their database with an SQL data dump, whereas a Laravel developer uses Seeds to input default data into their application.

If you were to build an application that does not have any data, it would look pretty useless. Luckily we can leverage *seeds*, which allow us to add default data into our database schemas. So, when we hand over our app to another developer we can always make sure there is a little bit of seed data to work with.

Seeds

Laravel allows us to add test data into our application. This is referred to as seeding the database. In the previous chapter we talked about migrations and in this chapter we will talk about seeds which is the data stored in our database.

Let's work off of our previous examples where we had a zombie table in the previous chapters. Inside of the database we had 2 test zombies that were:

- Johnny Bullet Holes
- Ted Manwalking

Now if we have the database migration for the zombies table in the previous example we could hand over our application to another developer and they can run `php artisan migrate` and be up and running with the zombies table. Well, what if we also wanted to include our two zombies in the database?

We can create a Zombie Table Seeder for that.

Let's go ahead and use our friend artisan again:

```
$ php artisan make:seeder ZombieTableSeeder
```

After running this command we will see a new file has been created inside of `database\seeds\ZombieTableSeeder.php`, and the contents of that file will look similar to the following:

```php
<?php

use Illuminate\Database\Seeder;

class ZombieTableSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        //
    }
}
```

The function that will run when we want to seed the database is the `run()` method, and this is where we will want to put our code. To seed our zombie database with our two zombies we could add this to our `run` method:

```php
public function run()
{
    $zombies = array(
        ['name' => 'Johnny Bullet Holes', 'strength' => 'Strong', 'health' =\
> 70],
        ['name' => 'Ted Manwalking', 'strength' => 'Weak', 'health' => 90]);
    DB::table('zombies')->insert( $zombies );
}
```

What we have done is create an array of zombies containing their name, strength, and health. We then use the DB facade class to insert our zombies into the Zombies table. To run this seeder we will run the following artisan command:

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

And now if we look in our database we will have our 2 zombies in the Zombies table.

Now we can use version control for our data, so when a new developer pulls a fresh copy of our app on their computer they will also have the data from our database.

Gone are the days of passing around an `.sql` file and now are the days of creating migrations and seeds. This is just another example of how Laravel makes our lives easier. Let’s move on to learning about the built-in security that laravel provides.

Chapter 19 - Security

A zombie developer doesn’t care about security, whereas a Laravel developer can rest assure that their app is secure from some of the most common app vulnerabilities.

We want to make sure that no one can hack into our application, and luckily Laravel has been built to prevent some of the more common ways that users hack into systems.

Security

Security is an important thing when it comes to building applications. Luckily for us, we have decided to use Laravel which includes security features such as SQL Injection, Cross Site Scripting, and Cross Site Request Forgery.

Don't worry if you don't know much about any of these protections. We'll explain them below:

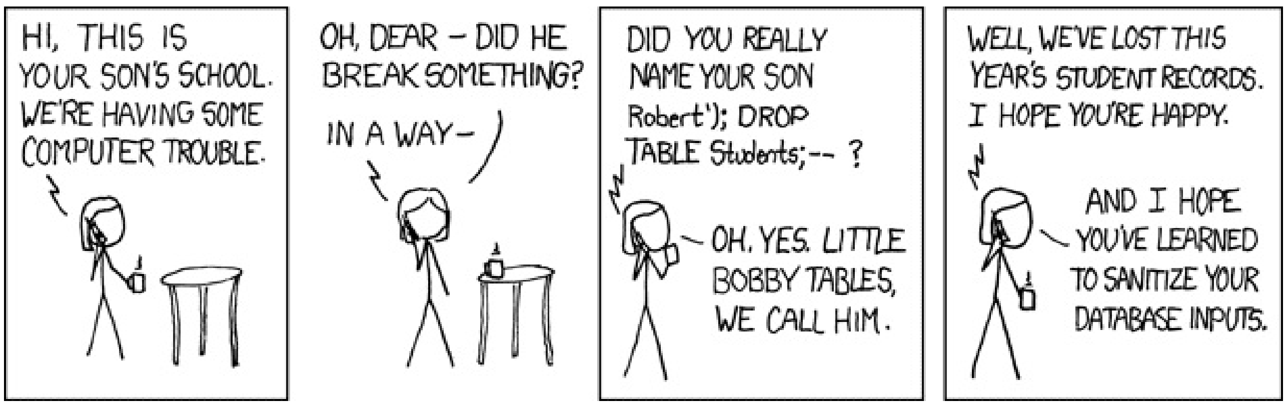
SQL Injection

SQL Injection is when someone tries to hack an input that gets submitted into the database. Say we were to run a SQL command like so:

```
$weapon_name = $_POST['weapon_name'];
$query = 'INSERT INTO weapons VALUES ('1', $weapon_name);
```

The user could easily enter in a value to the weapon name to Inject SQL into our query. So, they could potentially run a query and drop a table or a database.

Check out this XKCD.com comic:


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Thanks to Laravel and Eloquent we don't have to worry about SQL injection.

Cross Site Scripting

Cross-site scripting occurs when a hacker adds malicious code in the form of a client side script. So, pretend you have a comment text area and someone put in the following and submitted it as a comment:

```
<script>alert('hello, I just hacked this page');</script>
```

Now whenever someone visits that page, they will be alerted with this annoying popup. You can see that this could be dangerous because the user could even redirect that page to another page.

Thankfully by using the blade templating engine we can output any data and protect against Cross-site Scripting attacks by using the triple curly brace syntax:

```
{{{ $user_comment }}}
```

That output above would be sanitized to prevent any Cross-site Scripting (also referred to as XSS attack).

Cross-site Request Forgery

Finally, there is another attack called Cross Site Request Forgery that Laravel can prevent against. This occurs someone modifies a POST/PUT/DELETE request being sent to your server.

A possible scenario is a hacker who modifies the data being sent by a request from the browser to the server. The hacker could intercept the request and swap out values, causing the web application to perform functionality that it normally might not have.

Thankfully, Laravel is here to save the day against CSRF attacks.

When you submit a form you can include a hidden input type with a name of _token and the value of `csrf_token()` and Laravel will handle the rest. The form will look similar to the following:

```
<form method="POST" action="/zombie">
    ...
    <input type="hidden" name="_token" value="{{ csrf_token() }}">
    <input type="submit">
</form>
```

And Laravel will handle all the rest. If a POST/PUT/DELETE request is submitted and the security token does not match, the data will not be posted to the application.

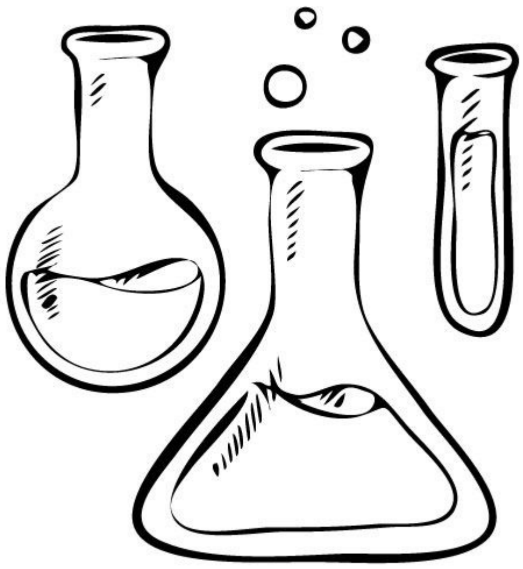
Note: To leverage this CSRF protection you will need to use the `web` middleware group that we talked about in previous chapters:

```
Route::group(['middleware' => ['web']], function () {
    // Your routes here will use CSRF protection
});
```

And you can sleep peacefully knowing that your site is safe against any CSRF attacks.

It can get very tiring making sure your app is as secure as possible, but thanks to Laravel we can focus on what we enjoy most, building our app.

Chapter 20 - Testing



A zombie developer releases code and hopes it doesn't break, whereas a Laravel developer writes automated tests to guarantee that new code does not break any functionality in their app.

Say that you are given a few grenades during the zombie apocalypse and the person giving you these grenades says, "I think they should work, they've been stored away for many years". Wouldn't you rather have them say, "These are our finest top of the line, tested to blow the roof off of anything grenades"

Yeah, of course you would feel better throwing the grenade into a swarm full of zombies that you know are tested to work. That's why testing is so important. We want to guarantee that our app works in any situation.

Testing

Testing your app is essential for ensuring it works correctly.

I'm sure we've all done some testing to some extent. If we open our application, look at data, or even click a few links then we have tested our application. The only problem with manual testing is that it can be very time consuming.

Imagine for every line of code you change you have to go back and run through your whole application to make sure it's all functioning correctly. That would be absurd, right? Only a zombie would mindlessly perform these repetitive tasks over and over again.

Lucky for us we can automate our testing by using PHPUnit that is included by default with a fresh install of Laravel.

Let's go over an easy example of how testing can help us out. Let's say that we have a page with a simple link called 'Invetory of Weapons' that would bring us to a page that says 'Weapons.'

Well, in that case, we would probably have a view file called `artillery.blade.php` that contained the following HTML:

```html
<html>
<head>
     <title>Artillery</title>
</head>
<body>

     <a href="/weapons">Inventory of Weapons</a>

</body>
</html>
```

So, this page is loaded when we go to `site.com/artillery` and when that link is clicked we go to a page at `site.com/weapons`, so we would need 2 routes for this, which would look like the following:

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

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

And each of these will load the view. Now, we want to guarantee that anytime we visit the artillery page we see 'Inventory of Weapons' and when we click on the link, we then end up on the weapons page.

For simplicity sake we will just assume that when you land on the weapons page (located at `resources\views\weapons.blade.php`) that it has the text 'Weapons' on the page, like so:

```html
<html>
<head>
    <title>Weapons</title>
</head>
<body>

    <h1>Weapons</h1>

</body>
</html>
```

Next, lets create our first test.

```
$ php artisan make:test ArtilleryTest
```

After running the artisan command above, we will end up with the following code inside of a new file created at `tests\ArtilleryTest.php`:

```php
<?php

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

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

So, by default, we are given a test example. This test example is just hard-coded to be true. So, let’s run this test by typing in the following command:

```
$ vendor/bin/phpunit tests/ArtilleryTest
```

After running this command you should see a message in your command line that says something similar to:

```
OK (1 test, 1 assertion)
```

Which means that we have run 1 test and 1 assertion and everything was fine.

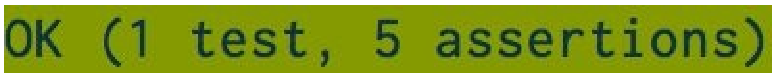
Why don’t we go ahead and add a new method to our ArtilleryTest class that looks like the following:

```php
public function testArtilleryPage(){
      $this->visit('/artillery')
          ->see('Inventory of Weapons')
           ->click('Inventory of Weapons')
```

```
            ->seePageIs('/weapons');
}
```

Notice that each function must be prepended with `test`. Let's run our test again:

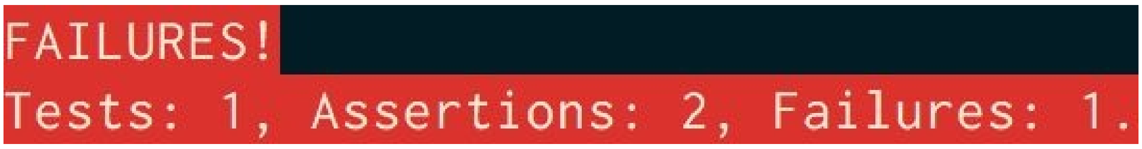
```
$ vendor/bin/phpunit tests/ArtilleryPage
```



```
OK (1 test, 5 assertions)
```

And we should see a green success message since we went to the artillery route and clicked the 'Inventory of Weapons' and then landed on the weapons page.

Try changing up the text. Say for instance that the link in the artillery.blade.php file said 'Inventory of our Weapons' instead of 'Inventory of Weapons'. If we run the test again, we will see a red error message saying that our tests have failed.



```
FAILURES!
Tests: 1, Assertions: 2, Failures: 1.
```

You can think of this as a game if you like. When we see green we are currently winning! But if we see red that means we have a problem and we need to figure out what needs to be fixed to pass our test.

Just imagine every time we make a modification to our code we could simply run through our tests and guarantee that we have not broken anything. How much easier would it be to sleep at night knowing that the code you just pushed didn't break anything?

This was a very simple example just to give you a quick idea of how tests can work, but there are many more things that you can test besides user interaction. Be sure to read up more about Laravel tests on the documentation page. We'll also provide some awesome resources you'll want to check out in the next chapter.

Chapter 21 - Wrapping Up

A zombie developer does not typically read books, whereas a Laravel developer finishes books they have started and they always continue learning.

It looks like you’re on your way to becoming an awesome Laravel Developer!

Wrapping Up

Whoa! We made it! You just learned the basics of Laravel, and shortly you’ll be on your way to building the latest and greatest app!

There are so many more fun things to learn about Laravel that we have not included in this book, so be sure to head over to the Laravel documentation and give it a read.

It’s one of the most enjoyable documentations available. http://laravel.com/docs

Resources

Laravel has quickly become one of the most popular PHP frameworks available today. It should be no doubt that there are plenty of resources out there that you will want to have in your back pocket.

Here are some resources that will help you on your way to advancing your Laravel skills:

- https://laravel.com/docs (The Laravel Docs)
- https://laracasts.com/ (Best Video Resource for Modern PHP and Laravel)
- https://laravel-news.com/ (Latest news on Laravel)
- https://scotch.io/ (Awesome resource for Laravel Articles)
- http://devdojo.com/ (Video resource with Dev & Laravel Screencasts)

Words of Encouragement

Before you set off on your journey to becoming a better PHP developer and leveraging the powers of Laravel I want to share a bit of wisdom with you.

Having fellow developers around you can help you grow and increase your skills; however, there may also be other developers who will tell you the way you are doing things are wrong and that you should be doing things this way or that way.

These kinds of people can distract or discourage you from time to time.

So, my words of encouragement when you encounter people like that is to take what they have to say with a grain of salt. Many people will help you learn something new, but some may prevent you from learning something new.

Never be afraid to take chances and to keep pushing the envelope.

My advice is to block out any negativity or discouragement. We are all in this together and when it comes down to it, there is no right or wrong way to do things. If you have done something wrong and it has forced you to learn, then that wrong thing was the right thing to do to get you to your next level.

A New Reality

Using Laravel and bettering your knowledge of development will open up a wide array of opportunities. A new reality will open up, whether that is to work for yourself one day or to make some extra money on the side.

Being a web developer is an excellent and exciting career choice. The demand for web developers is at an all time high.

If you are just starting off in web development, I would like to encourage you to keep on going! If you have already been doing web development for some time, give yourself a pat on the back because you’ve made a great choice.

Most people will agree that building or creating something is very gratifying, and as web developers we get to do just that! We get to take an idea and make it come to life!

The web development world is always evolving and constantly changing, so to continue growing as a web developer here are some key points that you’ll want to remember: keep learning, keep encouraging, and keep creating.