

A Laravel Prime

Getting started:

Laravel is a framework built on several other toolsets. As such, there are powerful tools to get your project up and running directly from the command line. Normally you would need to install composer in your computer before being able to follow this tutorial, if this is the case, you can google for composer and follow the installation instructions. In the labs in Block16 composer has already been installed.

Start by opening your terminal window on a mac, or the command prompt on windows.

You will want to keep records of your project as we build up from scratch, so it may be better to store the project on your P: drive. You should be able to change to this drive in windows using P: and then select an appropriate folder with cd in both windows and mac, however the mac path might be different than what is shown here. In windows command prompt type:

```
P: ↵
```

```
cd your-root-document-folder-for-COM409 ↵
```

Now you would normally run the composer tool to create a new project in Laravel using the latest version, by running the following command (do not do this yet!):

```
composer create-project laravel/laravel your-project-name --prefer-dist ↵
```

This would take a few minutes while packages are being downloaded. However, for the lab, this would probably trigger a security response from Github as the same project files get downloaded by your class mates. To avoid this situation, we will simply download a copy from our own GitLab space. Use this command:

```
git clone  
ssh://git@gitlab.scm.ulster.ac.uk:9922/COM409_6977/laravel_source.git ↵
```

This will download a zip file with a default empty installation of Laravel 5.2.

Once this is done, it's time to test the empty project. To do this, first unzip the file to a folder with your project name on the root of your COM409 folder in the P: drive and navigate inside the newly created project

```
cd your-project-name ↵
```

And then run:

```
php artisan serve ↵
```

At this point, the php web server has taken over control of the command prompt/terminal window. This is normal, as the php web server is running. You can now check that the page loads from your favourite web-browser by pointing it to `localhost:8000` (See figure). Notice that it uses a different port than the normal port 80 for http requests, this is by design and a security feature. Once your application is ready for a production environment you may need to set-up a virtual host pointing to the right folder in your project. More on this later. If you need to close the php web server, `ctrl-c` in the command line/terminal window will stop it.

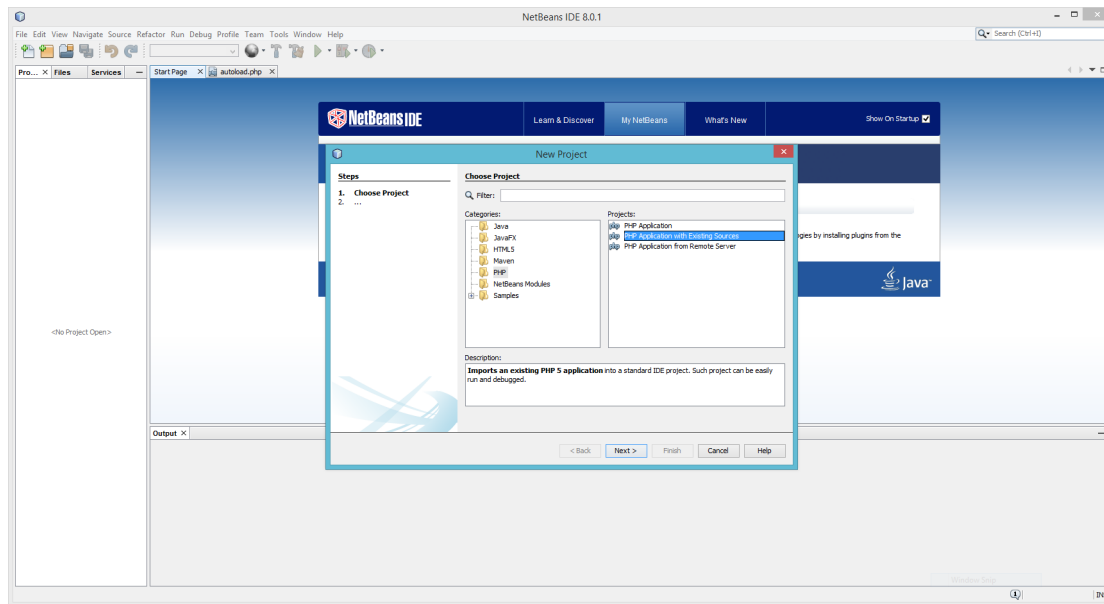
In your browser, you should see the “Laravel 5” message. This means your project framework is ready to be used.



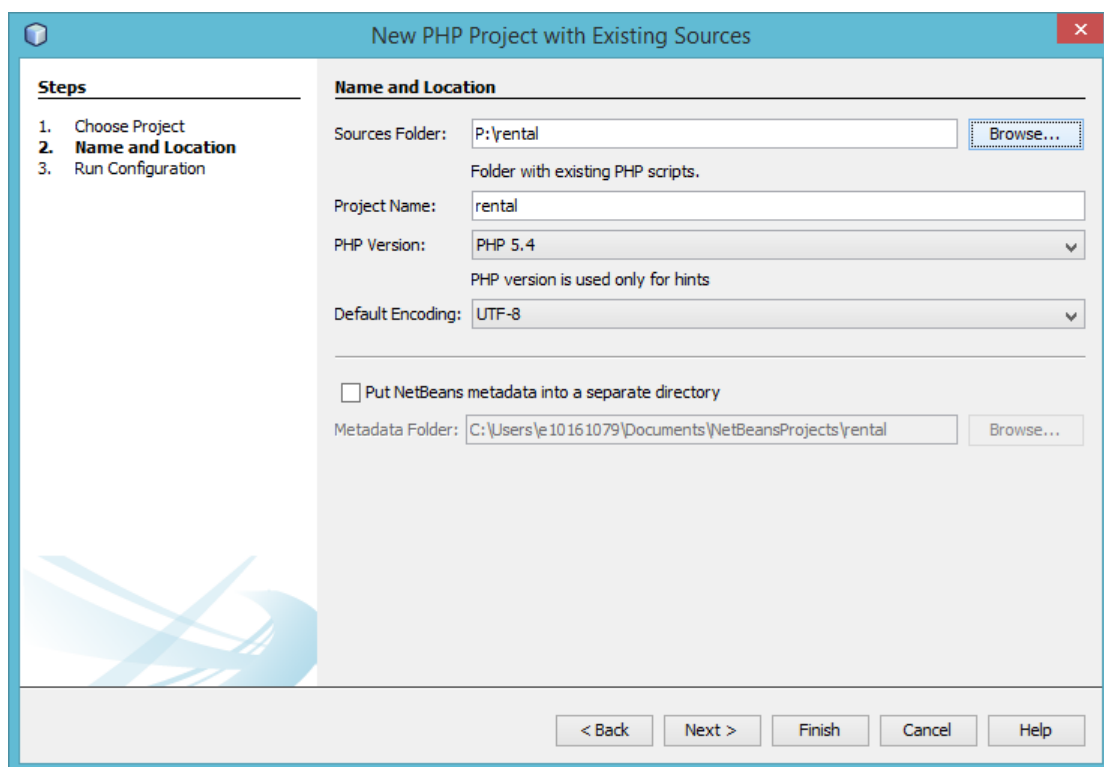
Now you should set-up Netbeans (or your preferred editor) to manage your project easily. Netbeans has an already familiar interface for you, but there are other great editors you could try: working in both windows and mac are phpStorm*, Sublime text, or VIM; in a mac you can also try TextWrangler or Coda; in windows you can also continue to use Notepad++.

Using Netbeans to manage your coding

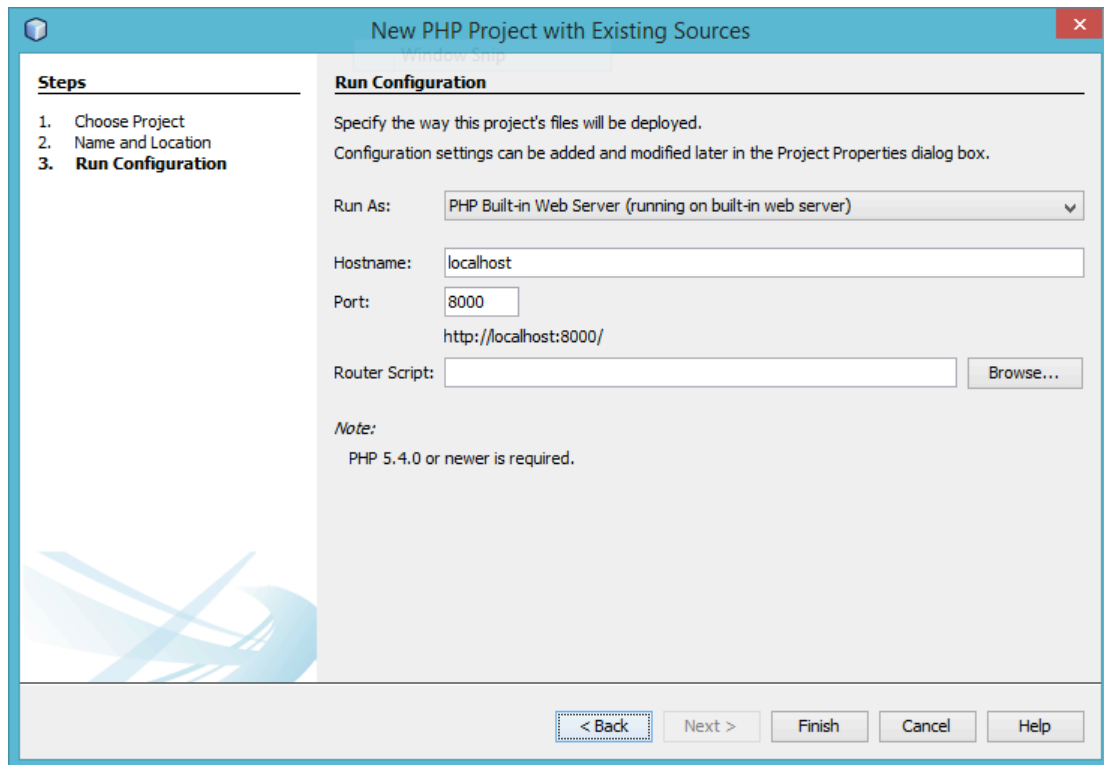
Open Netbeans, and select New Project from the menu and “new PHP project with existing resources” from the quickstart screen and click next:



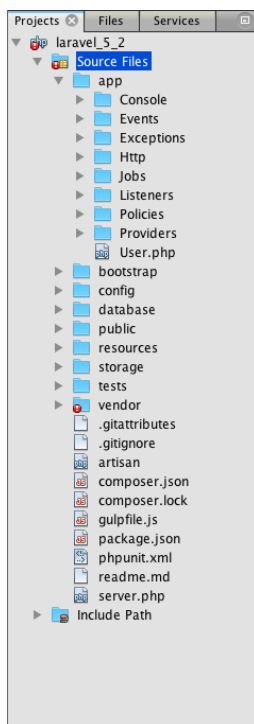
In the next screen, fill all the relevant information for your own project and click next, see sample below:



In the next screen set-up your starting script for the project selecting PHP built-in Web Server with localhost as the hostname, and 8000 as the port, as shown below.



Leave the router script empty.



You should now have the familiar netbeans interface you normally use to develop your Java projects. On the left side, you should see the standard navigation frame.

Here you can see the folder structure of your default Laravel Application. Inside the “Source Files” folder you have your App folder, which is where most of your changes will live, bootstrap folder, which controls the start-up of your project, the public folder, which is the only folder that needs to be visible for your web server, and a vendor folder, which is where you will place all the different 3rd party plug-ins and tools you may require while building your application.

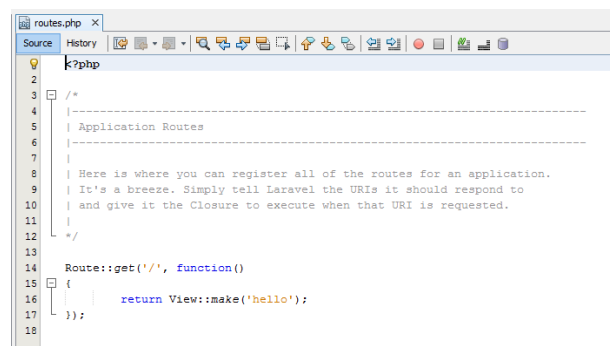
Shown in the image is the app folder expanded. Here you can see some folders, but one of these folders is particularly important: Http. Inside this folder you will find a Controllers folder, a Requests folder and a particularly important file: routes.php.

The routes file has the responsibility of directing the flow of your application in Laravel, it will react to get and post requests as necessary, so it is no surprise that we need to understand this file in detail.

Routes

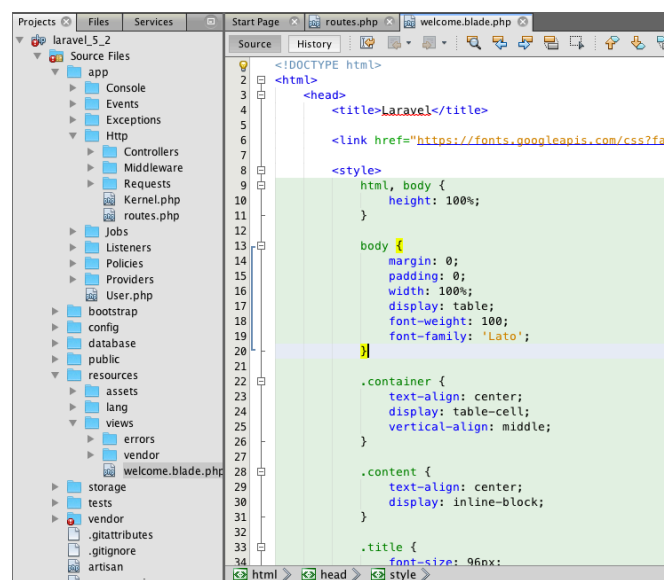
It is important to understand routes in Laravel, as they essentially control the way your application responds to a request from a web browser. A route can be understood as a traffic sign on a large motorway. It will direct traffic to its intended destination. The file will contain in essence all the information that your application requires to serve a request, starting from a simple “when this folder is requested, do this” architecture, to much more sophisticated ones using resources, which we will see later on.

If you open the routes file now, you should see the default route shown below:



```
1 k?php
2
3 /*
4  * Application Routes
5  */
6
7 Here is where you can register all of the routes for an application.
8 It's a breeze. Simply tell Laravel the URIs it should respond to
9 and give it the Closure to execute when that URI is requested.
10
11 */
12
13 Route::get('/', function()
14 {
15     return View::make('hello');
16 });
17
18
```

This file essentially tells Laravel that when a **get** request (one of our familiar type of requests) is received for the default index page (the '/') to run the immediate anonymous function (also called closure) following. This function only returns a View, named 'hello', which is found in the resources/views folder and is partially shown below:



```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <title>Laravel</title>
5
6 <link href="https://fonts.googleapis.com/css?family=
7
8 <style>
9     html, body {
10         height: 100%;
11     }
12
13     body {
14         margin: 0;
15         padding: 0;
16         width: 100%;
17         display: table;
18         font-weight: 100;
19         font-family: 'Lato';
20     }
21
22     .container {
23         text-align: center;
24         display: table-cell;
25         vertical-align: middle;
26     }
27
28     .content {
29         text-align: center;
30         display: inline-block;
31     }
32
33     .title {
34         font-size: 96px;
35     }
36
```

Notice that the View::make method in the routes file takes only the name of the view file, without the .php extension.

Task 1

In order to check that this is indeed the main “sign” in our highway, let’s edit this file to simply return a string message, rather than a full view. Make the following changes to the file:

```
Route::get('/', function () {  
    //return view('welcome');  
    return "Hi, you have just changed the routes file";  
});
```

Save the file and refresh the page in your web browser (or run the project in Netbeans). You should see that the beautiful Laravel 5 screen has been replaced with your ugly message.

Task 2

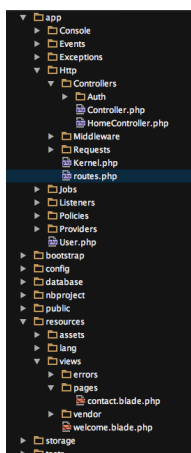
Let’s build another route. This time, we want to point our browser to localhost:8000/contact and see a new page with some contact information in it. Try to modify the routes file appropriately to add a new route, pointing to ‘contact’ and the closure function should return some sensible text appropriate for the contact page.

Save the file and test the page in your web browser. Seek help at this stage if you are not able to see your contact page.

Task 3

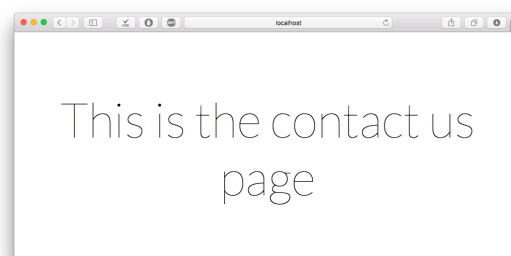
This time, we want to point our browser to localhost:8000/contact and load a page from a view (like the home page was showing at the start) Modify the routes file appropriately to change our route so that now the closure function returns a view called contact. You will need to create the view as well, but you can effectively duplicate the welcome view and modify it to your liking.

Save the file and test the page in your web browser. Seek help at this stage if you are not able to see your contact page.



Here’s my version of this, see if you can figure out what is going on.

```
Route::get('contact',function () {  
    return view('pages.contact');  
});
```



Passing Data to a view

So far we have been playing with routes but with pretty static information. Most projects will have a number of these static pages, but most of the application will depend on user input and data sources. So, how do we work with these in Laravel?

Let's start by creating an array of data in our route. Let's take the welcome page route and as part of the response, we will generate an array of people associated with Laravel, like this:

```
Route::get('/', function(){
    $people = ['Taylor', 'Matt', 'Jeffrey'];
```

Next, we want to pass this information to the view. There are several ways to do this in Laravel, and they are all equivalent. Choose the one you like the best, and be consistent:

```
• return view('welcome', ['people'=>$people]);
• return view('welcome', compact('people'));
• return view('welcome')->with('people', $people);
• return view('welcome')->withPeople($people);
• return View::make('welcome')->withPeople($people);
```

The end result is the same. A new array will be created with the values for \$people associated with it, and passed on to the view 'welcome'.

How do we use these values then on the view? Well, Laravel provides again different ways to access these values. Let's start with just some classic php to address this. Change the welcome.blade.php file so that it looks like this:

```
<body>
  <div class="container">
    <div class="content">
      <?php foreach ($people as $person){?>
        <li><?php echo $person; ?></li>
      <?php } ?>
    </div>
  </div>
</body>
```

Test this in your browser. Notice how having the PHP code embedded in the html looks a bit gross. Yes, PHP started as a templating engine, but it didn't evolve to be a very good one. Instead, other options are available. Laravel ships with its own templating engine called Blade. Let's see how that would look like:

```
<body>
  <div class="container">
    <div class="content">
      @foreach ($people as $person)
        <li>{{ $person }}</li>
      @endforeach
    </div>
  </div>
</body>
```

Notice that this is effectively the same code, but it looks much more elegant and readable. Try it yourself and convince yourself that this works.

Task 4

What would happen if the people array is empty? Try this in your code. Try to fix the issue, either by using PHP code or attempt to fix it using Blade's own syntax.

Controllers

Laravel comes by default with a "User" model, and an empty controller already set for you. Most new controllers you create however will follow the same idea and extend from this Controller.

Controllers are a very convenient way of grouping related tasks in the one place. And in combination with routes can provide a very powerful way to direct your requests. For example, you could group all your normal User Interaction tasks within a UserController. This controller will deal with all post and get requests done to the root level of your application, and focusing on great formatting of your data for human consumption. Then you could have another Controller, for example an ApiController, which will deal with all calls dealing with raw, unformatted data, for developers and programs consumption using the REST paradigm. You could have a Controller to deal exclusively with Admin requests, another one for Registered Users, etc.

Typically, a controller will deal with a type of data, and certain views. So it will rely heavily in the Model files and the View files. We will explore some of these later.

Building a controller is a task that you will perform a lot during the development of an application. Instead of manually creating a controller, Laravel provides us with ways to speedup this task. In your terminal window, you can take advantage of Laravel command line interpreter, called artisan, to generate all sort of helper code. Let's see how we can generate a controller for our application. In the command line type:

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

Then open the new file that has been created inside the controllers folder. You will see that the file has been created with minimum content, but it is ready to use. The namespace has been appropriately set, and a number of support classes have been "imported" to the file. Further, the class PagesController has been created and is ready to be populated by you.

Task 5

Let's make use of this new shiny controller we have created! First of all, we need to modify our routes file to use our new controller:

```
//Route::get('/', function(){  
//    $people = ['Taylor','Matt','Jeoffrey'];  
//    return View::make('welcome')->withPeople($people);  
//});  
Route::get('/', 'PagesController@home');
```

I have commented the old code for now, because we will still use some of it. Note how we are calling the new PagesController and we are asking for an “action” named home. This will be a method of the new class. So let's create that. In the PagesController file add this code:

```
class PagesController extends Controller  
{  
    public function home(){  
        $people = ['Taylor','Matt','Jeffrey'];  
        return View::make('welcome')->withPeople($people);  
    }  
}
```

Save the files and reload the home page in your browser, and you should see the previous welcome screen.

Try to do the same for the contact page, without my help.

Adding Stylesheets and Scripts– Using Laravel helper functions

Your application may require some common files that would typically be served from the “public” folder in a web server. A typical example is the cascading style sheets that you may associate with your application. You want these files to be visible for the browser and accessible from any view, in the public folder. But since your views live in a different folder, away from your web server's vigilant eye, you need a way to add it so that it doesn't break the application or your security.

Enter Blade

Blade is Laravel's default templating engine. It is slick, easy and quick to use and comes installed with Laravel by default, so there is no real reason not to use it. Blade makes using code in HTML a painless task, and the syntax is very easy to learn.

Once Laravel knows you are using the Blade template, you can start making full use of its power. For starters, Blade makes it easy to switch between template and data with the use of the double { as we saw previously. But there is more to Blade than just parsing data. For example, you could create a “head” partial, and a “foot” partial, which would contain all the code that must remain the same in the header and footer of your pages respectively. And then you can add these partials to the actual pages. This way, if in the future you need to change a copyright notice at the foot of every page, you only need to change one file. This is done with the use of layouts.

Using Layouts

You can think of layouts as blueprints of how you want your page to look. You build these with the knowledge that certain areas of the page will be content-dependent. Layouts in Blade can be very powerful, because you can set a standard look, but each page based on that look could also modify certain functionality, as necessary, thus ensuring your site looks consistent but also being flexible when you require.

Task 6

Let’s start by creating a folder inside your views, called layouts. Within this folder, create a file master.blade.php. This file will be our master template, so it must have most of our setting up HTML. Type the following lines into this file.

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge,
        chrome=1">
    <title>My Website</title>
    <meta name="viewport" content="width=device-width">
</head>
<body>
    <div class="container">
        <!-- Start main section -->

        </div> <!-- End of container -->
</body>
</html>
```

This is standard best practice on HTML to use HTML5. We have set-up some information so that the browser uses the latest rendering engines. We set up a title for our pages and started our body with a main div of class container where we expect our page to be defined. So, how do we attach this master template to our page? We need to tell Blade to use this template, and we do so with the keyword @layout.

Modify your welcome.blade.php file, so that it uses the master layout we have just created. To do so, open the file, and enter the code:

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

at the start of your file. If you try this now in your browser you will probably not see any difference (or an error message). You are using your layout, but we haven't told the layout how to display the specific page information. Let's do that next.

In your master.blade.php file, add this line inside the container div:

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

In your welcome.blade.php file, add this code after the @extends line:

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

```
    Here is the text that will be displayed in your webpage
```

```
@stop
```

You may replace the message with your own custom message from before.

Task 7:

Try to now add your css file to the master template, and use it from your welcome page.

In your home controller, you should now have a showHome() method or something similar which you created back in Task 5. Inside this method, create a variable \$title = 'A title', and a variable \$content='Some content'. Modify the return line so that it reads similar to this (use your preferred standard):

```
return View::make('welcome')->with('title', $title);
```

Now, to make use of this variable in your view, open the welcome.blade.php file, and inside the section "content" add this line:

```
The page title will be {{ $title }}
```

Now load the page in your browser. You should see that the correct text is displayed. We saw before that there are other methods to pass the data to a view. You can also use php magic methods to create variables in a view, like so:

```
$view = View::make('welcome');  
$view->title = 'A Title';  
$view->content='Some content';
```

```
return $view;
```

Exercise

Pass and display two variables, to your view, displaying them in the right section (for example, title in the header section of your page, and content in the body).

Exercise

Try to create a few un-linked pages following what you already know of Laravel and Blade, maintaining a common css and structure. Now you can focus on making things look pretty! Don't worry just yet about making the pages link to one another. But make sure you create a route to the page, a method in your controller for each one and a corresponding view with some "dynamic" data (ie, data you can modify in your controller and gets passed to the views).

My solution to task 4

```
<body>
  <div class="container">
    <div class="content">
      @if(empty($people))
        There are no people
      @endif
      @foreach ($people as $person)
        <li>{{ $person }}</li>
      @endforeach
    </div>
  </div>
</body>
```