Hyperiondev

TASK

PHP -

Laravel Framework

Visit our website

Introduction

PHP WITH A JETPACK

If you think using PHP makes web development easy, wait until you hear about this! Laravel is one of the most popular web app frameworks for PHP. In past lessons, you learned about the nitty-gritties of PHP — from parsing POST data to managing multiple webpages to manually setting cookies. Laravel takes care of all these fundamentals allowing you to focus only on the higher-level aspects of making your web app. In this task, we're covering how to use Laravel.

SETTING UP LARAVEL

The first thing that needs to be done is to set up Composer. Composer is a dependency manager — much like what npm is to JavaScript, or what Maven is to Java, it takes care of managing other people's libraries so that you can easily pull a library in and build on top of it. Laravel is one such library we'll be using. An installation guide for Composer is available **here**.

After installing Composer you can install Laravel with the following command from a terminal. It will download and make Laravel's libraries and functionality available on your system so that you can use it to create Laravel projects.

```
composer global require laravel/installer
```

Laravel requires a certain list of PHP extensions to work, most of which are enabled by default. If you're missing such a PHP extension, the above install command will let you know. A quick Google search should put you right on track for enabling the missing extension before restarting the Laravel install. The most common extension that will block this installation is the **fileinfo** extension. You can enable extensions in php.ini the same way you did for mysqli in the previous task by removing the semicolon that prefixes the extension. Also, ensure that the **pdo_mysql**, **mb_string** and open_ssl extensions are also enabled.

Once you have completed these steps. It is important to restart your PC so that all extensions work as expected.

To create a Laravel project, run the following command. We'll be creating a basic website, so we called it "cool-blog".

```
composer create-project --prefer-dist laravel/laravel cool-blog
```

This will create a basic project in a directory called cool-blog. It will also install the final dependencies Laravel needs to run your web app. If it detects a missing PHP extension it will stop and let you know. In this case, enable the missing extension, delete the directory, and run the command again.

After it has completed successfully, `cd` into the project directory and run the following to start a local webserver.

```
php artisan serve
```

Now head over to `localhost:8000` in your browser (or in Eclipse) to see the basic website up and running.

For a lot of Laravel's functionality, it needs access to its own database. You should have a MySQL Server running on your machine from previous tasks. Get to a MySQL command line and run the following to create a database and a user for Laravel.

```
CREATE DATABASE laravel;
CREATE USER 'laravel'@'localhost' identified by 'Laravel123';
GRANT ALL ON laravel.* TO 'laravel'@'localhost';
```

Open your root project directory in your IDE of choice. Open the `.env` file and change the following properties to inform Laravel how to connect to the database:

```
DB_CONNECTION=mysql
DB_HOST=localhost
DB_PORT=3360
DB_DATABASE=laravel
DB_USERNAME=laravel
DB_PASSWORD=Laravel123
```

ROUTING BASICS

When checking out the contents of your project directory, you'll notice a lot of stuff. There is pre-populated basic content for bootstrap, app configuration, static files, web routing, server-side storage, unit testing, and much more. It's everything you

need to build a fully-fledged web application. We'll only be focussing on a subset of what is available, in order to get a very basic web app working.

In your IDE, open the file **web.php**, in the folder **routes**. This file specifies what your user will see when browsing different URLs on your website — web routing. Unlike with pure PHP where a file is a web page, in Laravel, a file can contain content for multiple web pages, or a webpage can be made from multiple files. Everything is automatically managed for you.

You should see a route specification for the URL **'/'**. This is known as the root or base URL, and it is what a user sees by default when browsing to a website. It's what is meant by the final slash in a simple URL, e.g. **https://www.google.com/**. Anything after that slash indicates different possible pages on your website. E.g. **google.com/mail**, or **google.com/images**.

The **get** part of the expression indicates that the route is only valid for GET requests. You can specify another route by using **post()** to accept POST requests. The second argument to the method is a function that is responsible for generating the route's response. This function (as we'll see later) can also take parameters to define what the response should be.

You'll see the route is written to return a view called "welcome". In Laravel a view is a renderable page. You can pass content into a view to have it render differently. We'll look at this later. For now, change the route spec to simply say "Hello World!".

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

Refresh your browser to see the change. Below this route spec, create a GET route for **/about** that returns just a few sentences about the nature of the website (blog). You can also use html tags in your response, but don't get too carried away. We'll look at generating custom views next.

VIEW BASICS

In Laravel, a view is like a web page template. It contains the bulk of a page's styling and content, but certain parts of it can be dynamic — i.e. different per render. We'll be creating one such view now. In the directory **resources/views**, create a file

called **home.blade.php**. A **blade.php** file is an HTML file, but with custom syntax for integrating PHP code. Place the following in your newly created file:

```
<html>
<head>
   <title>Cool Blog Homepage</title>
</head>
<body>
   <h1>Welcome to the homepage of my cool blog!</h1>
   <a href="{{ url('/about') }}">About</a>

   <p>Today is {{ date('j F Y') }}.</p>
</body>
</html>
```

Anything between double-curlies is PHP code. This is a similar tactic to many JavaScript frameworks, but a key difference is that Laravel evaluates server-side, while JS frameworks evaluate client-side. This means Laravel has quick and easy access to server resources while generating responses — later we'll see why this is useful.

The first double-curly expression calls a function called `url`. It is a Laravel (not PHP) built-in function that generates a full URL pointing to a specific route in your web app. In this case, the about page you made earlier.

The second double-curly generates the current date.

Head over to `web.php` and change the `'/'` route to render the view we just made.

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

Refresh your browser to see your new homepage, and check that the About page link works. Create a proper view for your About page (called `about.blade.php`) and make the appropriate route render it.

CUSTOM VIEWING AND ROUTING

Sometimes you may want to render a page differently depending on certain input. For instance, you may want admin users to see a different settings page to regular

users. Before we get into conditional rendering, let's look at passing in input to generate custom views. Create a view called "greeting", and place this inside it:

```
<html>
<body>
   <h1>Good day, {{ $name }}!</h1>
</body>
</html>
```

By default, the `$name` variable has no meaning and so will evaluate to nothing. When calling the view to be rendered, we can pass in a value for `$name`. Let's do this now in its own route definition:

```
Route::get('/greet', function (){
  return view('greeting', ['name'=>'Kabelo']);
});
```

The second parameter to the `view()` function is an associative array. It acts like a map and is the same data structure as all the superglobals you've seen so far. Laravel automatically parses it and sets the appropriate variables when the view is rendered. Browse to `localhost:8000/greet` to see the final product.

We can also get input data from the URL itself. Update the route spec to take a required parameter in the URL:

```
Route::get('/greet/{name}', function ($name){
  return view('greeting', ['name'=>$name]);
});
```

The routing function now takes as input a name specified in the URL and passes it directly to the view to be rendered. Browse to `localhost:8000/greet/David` to see a new name be greeted. Try it with your own name.

CONDITIONAL RENDERING

For our example of conditional rendering we'll be using the legal page. Let's say we have two routes: `/legal/tos` and `/legal/privacy` which should house our terms of service and privacy policy, respectively. An appropriate view for both could look like this:

```
<?php
function pageName($ss){
   if ($ss === 'tos')
       return 'Terms of Service';
   else
       return 'Privacy Policy';
}
?>

<html>
<head>
   <title>{{pageName($subsection)}}</title>
</head>
<body>
   <h1>Legal: {{pageName($subsection)}}</h1>
   @if($subsection==='tos')
       <p>You may not access non-public areas of the website.</p>
       <!-- etc -->
   @else
       <p>Data collected by the website is not shared externally.</p>
       <!-- etc -->
   @endif
</body>
</html>
```

The first part of the file should be very familiar to you — normal PHP code! It defines a function that returns the page's title given the subsection part of the URL. That function is then called in the title part of the HTML and also in the heading of the page. Just below that, we have the syntax for blade template conditionals. It is equivalent to a normal PHP if-then-else switch, but with different syntax.

Let us now set up the route to define that **`$subsection`** variable.

```
Route::get('/legal/{subsection}', function($subsection){
   return view('legal', ['subsection'=>$subsection]);
});
```

It's good to also ensure only **`tos`** and **`privacy`** are valid subsections. We can do this by placing a regular expression constraint on the URL:

```
Route::get('/legal/{subsection}', function($subsection){
   return view('legal', ['subsection'=>$subsection]);
```

```
})->where('subsection', '(tos|privacy)');
```

`Route::get` creates a route object. One of its methods is `where()`, which allows us to define constraints for URL parameters. In this case we're constraining it to be either "tos" or "privacy". You can use **regular expressions** to constrain parameters in any way you wish. If you want to test it out, try restricting the `name` parameter of the `/greet` route to consist only of letters.

If a URL does not match the specified constraints, Laravel recognises it as an invalid request and returns with 404 (page not found). Try it out now by browsing to `/legal/conditions`.

DATABASE INTEGRATION

Laravel conveniently supports database integration. Specifically, you can easily make database calls using the Laravel framework and use the contents to render views.

First, let's create the appropriate tables. You can set up the table yourself with the SQL below. Remember to add some sample data for testing.

```
CREATE TABLE blog_posts (
  id INT AUTO_INCREMENT,
  title VARCHAR(255),
  content TEXT,
  created DATE DEFAULT (CURRENT_DATE),
  PRIMARY KEY (id)
);
```

To get started, we'll be listing all our blog posts on the home page. Alter the root route to get the database content and pass it into the view renderer:

```
// import the DB script. Add this to the other import statement at
// the top of the file.
use Illuminate\Support\Facades\DB;

// ...

Route::get('/', function () {
   $posts = DB::table('blog_posts')->get();
```

```
    return view('home', ['posts'=>$posts]);
});
```

In the first line of the routing function, all the items in the table "blog_posts" are retrieved and placed into the **$posts** variable. These are then passed into the home view, to be rendered.

Add the following to the body of your home view to list the post titles:

```
<h2>Blog posts:</h2>
@foreach($posts as $post)
    <p>{{$post->title}}</p>
@endforeach
```

The foreach blade imperative works exactly like the PHP foreach loop, but with different syntax. Thanks to Laravel's database wrapping capabilities, the post columns can conveniently be accessed like attributes on an object (thus **$post->title**).

Browse to **localhost:8000** to see the post titles listed!

If you get an error saying the database driver could not be found, it means the appropriate PHP extension is not enabled. Refer to the database setup part of this task for details.

Other cool things you can do with databases in Laravel:

```
// filter rows
DB::table('blog_posts')->where('title', 'LIKE', '%Coding%');
DB::table('blog_posts')->where('created', '>', '2019-01-01');

// get a specific record by id
DB::table('blog_posts')->find(3);

// get a specific record by search
DB::table('blog_posts')->where('title', '=', 'Hello World')->first();

// sort rows
DB::table('blog_posts')->orderBy('created');
```

This way of accessing the table is called using query builders, because the functions and parameters are used to eventually create SQL queries that Laravel runs on your behalf. Query building can also be used to update table contents, but we won't be covering that here. In many cases, query building functions can also be combined to create complex queries.

In this task we covered the most important parts of making a web app with Laravel and PHP. However, it's likely less than a tenth of what Laravel is capable of. For more information about the many wonderful features of the framework, check out their **comprehensive documentation**.

Compulsory Task 1

Extend the blog you've built so far to show actual blog posts.

- Create a route of the form `/blog/{id}` to view a specific blog post by id.
- Create a view that renders the title, content and date created of a blog post. Have the appropriate routing function render the view by passing in the appropriate record from the database.
- Update the home page blog post listings to include links to those specific posts.

Rate us

Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved or think we've done a good job?

Click here to share your thoughts anonymously.