php[architect]

APPROVED FOR PHP DEVELOPERS

May 2017
VOLUME 16
#200

Uncanny PHP

PHPARCH.COM

nexcess™

beyond hosting.

We’re hiring PHP developers

15 years of experience with

PHP Application Hosting

SUPPORT FOR *php7* SINCE DAY ONE

Contact **careers@nexcess.net** for more information.

The polyglot webtech conference on a cruise ship!
Leaving from New Orleans and sailing the Caribbean.

July 16-23, 2017 — *Tickets $295*

www.codercruise.com

Sponsored by:

CISCO DEVNET

When I Work · Roave · CIOReview · VPSSERVER.com · Engine Yard · stickermule

Drupal Watchdog · ActiveState · concrete5 · php[architect] · contentful · GitHub

salesforce · phpwomen · OmniTI · OSMI · INFINITE RED

MAY 2017
Volume 16 - Issue 5

Uncanny PHP

Features

Columns

Editor-in-Chief: Oscar Merida

Editor: Kara Ferguson

Subscriptions
Print, digital, and corporate subscriptions are available. Visit https://www.phparch.com/magazine to subscribe or email contact@phparch.com for more information.

Advertising
To learn about advertising and receive the full prospectus, contact us at ads@phparch.com today!

Managing Partners
Kevin Bruce, Oscar Merida, Sandy Smith

php[**architect**] is published twelve times a year by:
musketeers.me, LLC
201 Adams Avenue
Alexandria, VA 22301, USA

Contact Information:
General mailbox: contact@phparch.com
Editorial: editors@phparch.com

Print ISSN 1709-7169
Digital ISSN 2375-3544

Project Creation

Joe Ferguson

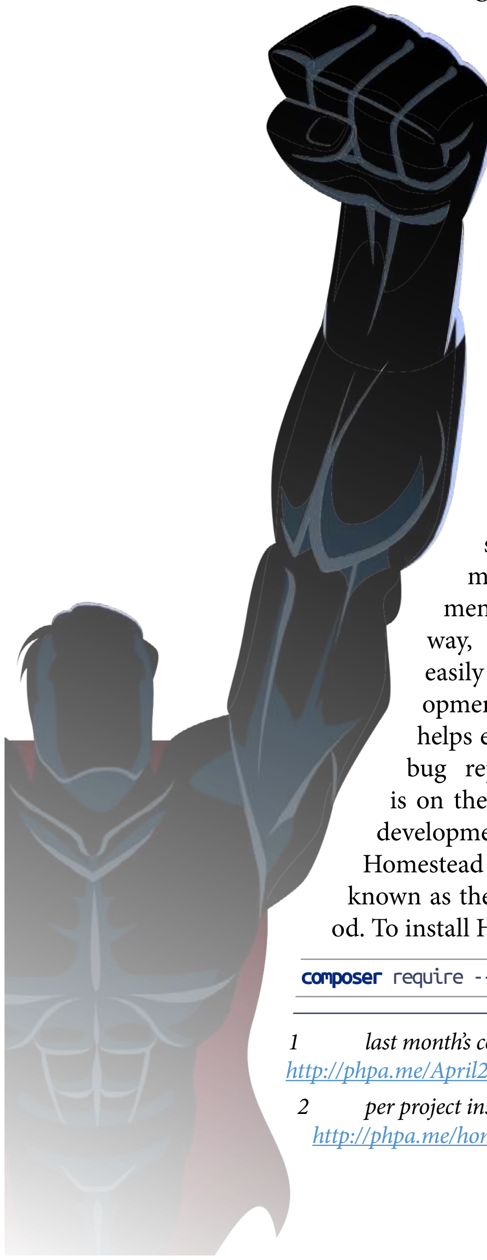
Every developer has a set way of starting a new project. Most frameworks have a linear path to getting started, and Laravel is no exception. With a few commands, you can quickly get started configuring routes, writing controllers, and saving data in a database.

You can install Laravel with Composer by:

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

My preferred way to create a new project is by using the Laravel installer. You can get the installer by adding it to your system with Composer:

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



Once installed you can use it to create a new project in a directory you specify. `larval new project` will create a new folder called `project` and install the latest version of Laravel.

The second command I run on all of my Laravel projects adds Homestead. As you read in last month's column[1] Homestead is my default local development environment. I install Homestead slightly different than many people. Often, I am working on projects I intend to open source later, so I always bundle my local development environment with my project files. This way, I know my contributors can easily spin up the same local development environment I am using. This helps eliminate many false errors and bug reports, and ensures everyone is on the same playing field during the development of the application. I install Homestead via Composer. This is also known as the per project installation[2] method. To install Homestead via Composer run:

```
composer require --dev laravel/homestead
```

1 *last month's column:* http://phpa.me/April2017issue

2 *per project installation:* http://phpa.me/homestead-per-project

One of the reasons I enjoy working with Laravel is its great developer experience. As web developers, we are often worried about the user experience (as we should be). Laravel always impresses me with the focus on making my job as a developer easier. Most full stack frameworks came about from the same reason: a developer was tired of writing boring boilerplate code for every project and decided to bootstrap pieces together. The power of packages and Composer allows that to happen even easier than it did years ago. Laravel allows developers to jump right into the business logic, the fun stuff of the application they're building.

Artisanal Frontend Development

My least favorite part of web development is frontend. I have never had an eye for design, and while I have developed a sense of what to look for in creating valuable user experiences, I do not enjoy it. I do like JavaScript, but I don't enjoy the rate of change of the frontend ecosystem. This is one of my favorite parts of Laravel: Laravel Mix. Mix is a collection of packages and pre-built workflows that take the pain out of nearly all the frontend bootstrapping of your application. Mix will give you Bootstrap, jQuery, and Vue support right out of the box. Coming in Laravel 5.5 there are even helper commands which will configure a Vue.js frontend or ReactJS. Laravel Mix removes a lot of the frustration of figuring how to do something today when it comes to frontend development.

Laravel 5.4 comes ready to work with Vue.js via Laravel Mix. All you need to get started is to run:

```
npm install
npm run dev
```

Alternatively, you can also use `yarn` instead of `npm` if you prefer. These two commands will install all the front end dependencies to the typical `node_modules` folder in your project and compile all the source assets from the `resources/assets` folder into compiled files in your `public/` folder. You will use the compiled versions of the CSS and JavaScript files in your templates and layout views. When you deploy to development, you will want to run `npm run production` so a few extra steps are taken to minify and optimize your files for production to ensure the best possible frontend performance of your application.

PHP[WORLD] 2017

This is a conference like no other. Designed to bring together all the communities linked by the PHP programming language.

Together as the PHP community, the sum is greater than the whole.

November 15-16, 2017
Washington, D.C.

world.phparch.com

Artisanal Testing

Once I have my development environment and frontend layout work configured, I start writing tests. I'm not a Test Driven Development (TDD) evangelist by any means. However, I know the value of tests and often find it easier to start writing basic tests I know will fail and then begin to work out the logic to make the tests pass. Some of you that are TDD veterans will know the Red, Green, Refactor mantra; I really like that approach.

Laravel Dusk is the newest testing package on the Laravel block, and it really takes the already impressive ease of testing to an entirely new level. Dusk utilizes the ChromeDriver[3] package to run tests in a real Chrome browser. When you run your Dusk test suite, you'll see Chrome open browser windows and execute the tests in front of you. It will send detailed error reports when something goes wrong. You can also run these tests from inside Homestead as of version 5.2.1. Obviously, because Dusk is using a real browser, these tests will take longer to process. Also, keep in mind these are *not* unit tests. Laravel Dusk tests are acceptance tests. Acceptance testing is often used to verify a product is working as expected. The tradeoff of using acceptance tests is they are slower and often don't show exactly where the problem may be. Acceptance tests are *not* a replacement for unit tests. They should be used together to ensure you have high test coverage of your application so when a bug does show up it is easier to find and fix.

One downside to Laravel Dusk is there is no way to reset the database state after every run. Laravel 5.3 test helpers (now known as `BrowserKitTesting`) could use Traits that would reset the database after a test had completed, meaning you were free to do what you wanted with the database and it would be reset back the state it was before the tests ran. When I do acceptance testing, I like to test everything going in and out of the database is as I expect. This requires resetting the database after each run. You can manually seed a second test database with an SQL file import each time Dusk runs. This is how I've managed to utilize the power of Dusk's browser testing and still not have to worry about altering my database.

As mentioned, the Laravel 5.3 testing helper functionality has been renamed to a `BrowserKitTesting` package you can easily add to your project via:

```
composer require —dev laravel/browser-kit-testing
```

You will then need to update your `composer.json` as in Listing 1.

Listing 1

```
"autoload-dev": {
    "classmap": [
        "tests/TestCase.php",
        "tests/BrowserKitTestCase.php"
    ],
    "psr-4": {
        "Tests\\": "tests/"
    }
},
```

Now you can write the Laravel 5.3 style tests. Make sure your tests extend the `BrowserKitTestCase` class. You can also copy tests from Laravel 5.3 to 5.4 applications with this method.

The advantage of using the BrowserKit Testing package is you can easily write functional tests which do whatever you want to the database and easily clean up after themselves. Since they're not using the ChromeDriver, they also run slightly faster than the Dusk tests.

Artisanal Routing

Laravel 5.4 has four files in the `routes` folder to handle routes. If you are new to Laravel, you should only worry about `web.php` and `api.php` since these are the HTTP routes and API routes respectively. Once you get farther down the path with Laravel and need event broadcasting, you'll refer to the `broadcast.php`. Likewise you may never need `console.php` unless your application is leveraging several complex Artisan commands.

Laravel routing syntax is very expressive and easy to read (just like the entire framework). The example route can be seen here:

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

This is a simple GET route which returns a callback. This callback returns a view named `welcome.blade.php`. Because we are using the view helper in our callback, we do not have to specify the full path to the views folder, nor the entire filename of the view. Routes can use any of the HTTP verbs such as GET, POST, PATCH, PUT, or DELETE.

Route parameters allow you to capture a part of the URI to pass into your callback. A common route you would expect to return a display view of a widget may look like:

```
Route::get('/widgets/{id}', function ($id) {
   $widget = Widget::find($id);

   return view('widgets.view')
         ->with('widget', $widget);
});
```

This route passes the ID value from the URI to the callback, and the callback attempts to find the widget matching the ID from the URI with an ID in the database. If a match is found, the row is returned and passed to the widgets view to be rendered.

Naming routes is the easiest way to keep your routes organized so a refactor down the road is even easier. The following

3 ChromeDriver: http://phpa.me/chrome-webdriver

is an example of a named route:

```
Route::get('/widgets/{id}', function ($id) {
   $widget = Widget::find($id);

   return view('widgets.view')
         ->with('widget', $widget);
})->name('widget.view');
```

This allows you to easily route to this function by its name, such as when you want to redirect a user:

```
// Return a redirect to widget.view route
return redirect()->route('widget.view');
```

If you ever needed to change the URI of this route you would only have to update the route in the routes file, not in many different places in your application.

When building an application that needs a management or administration control panel, you would normally group all of those routes behind an `/admin` route. Such as `/admin`, `/admin/widgets` or `/admin/tasks`. This can be easily accomplished by using route prefixing to keep your routes clean and readable.

```
Route::group(['prefix' => 'admin'], function () {
    Route::get('/', function ()    {
        // Matches The "/admin" URL
    });
    Route::get('widgets', function ()    {
        // Matches The "/admin/widgets" URL
    });
});
```

You can also assign middleware to routes. We mentioned creating a route prefix for grouping routes behind an `/admin` prefix. Assuming these routes allow administrative tasks it would make sense to use the `auth` middleware to secure these routes. The `auth` middleware simply requires the request to be from a user that has logged in. This keeps unauthenticated users from accessing any of these routes.

```
Route::group(
   ['prefix' => 'admin', 'middleware' => 'auth'],
   function () {
      Route::get('/', function () {
         // Matches The "/admin" URL
      });
   }
);
```

Adding Middleware to Route Groups

By now your routes file is filling up with a lot of things which don't need to be there, like all of those callbacks. Callbacks are great for small things and even for testing ideas for logic before committing them to any needed abstraction layers. I often keep a `/test` route in many of my applications for this very reason during development. However, once you're happy with the functionality, you'll want to move those callbacks to a controller method which prevents your routes file from becoming a hard to read thousand line monstrosity. This also allows a clean separation of duties and results in much easier to read code.

We can refactor our earlier route by creating a `WidgetsController` via the Artisan command:

```
php artisan make:controller WidgetsController
```

The Artisan command will create a basic boilerplate controller for us so we can just drop in our logic from the callback. Listing 2 goes in the `app/Http/Controllers` folder.

Since we have moved our callback into a controller method named `viewWidget` we can now update our route:

```
Route::get(
   '/widgets/{id}', 'WidgetController@viewWidget'
)->name('widgets.index');
```

Now we can clean up our routes file and start abstracting our callbacks into controller methods.

Cross-Site Request Forgery Protection

Laravel routing automatically handles Cross-Site Request Forgery (CSRF) checking. Any route that points to POST, PUT, or DELETE must have a CSRF token in the form data. You only have to ensure you use the view helper `{{ csrf_field() }}` in your form, no need to do anything on the server side processing. Laravel will automatically reject the request if the tokens do not match.

Artisanal Databases

Whether you are a die-hard Postgres or MySQL fan, Laravel has you covered. Laravel ships with support for just about any flavor of database being used by modern (and in some cases legacy) web development.

The basics of connecting your Laravel application to a database is to create a migration. If you have never worked with database migrations, they are simply PHP class files that tell an ORM (Object Relation Mapper)—Eloquent, in our case—what to do with a database schema. A migration has two methods: `up` and `down`. The `up` method is executed when

Listing 2

```
<?php

namespace App\Http\Controllers;

class WidgetController extends Controller
{
   public function viewWidget($id)
   {
      $widget = Widget::find($id);

      return view('widgets.view')
         ->with('widget', $widget);
   }
}
```

you run `php artisan migrate` and the `down` method is executed when you run `php artisan migrate`. Whatever you do in the `up` method should be reversed in the `down` method. An example migration to create our widgets table can be found here in Listing 3, found in the `database/migrations` folder.

Listing 3

```php
<?php

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

class CreateWidgetsTable extends Migration
{
   public function up() {
      Schema::create('widgets', function (Blueprint $table) {
         $table->increments('id');
         $table->string('name');
         $table->text('description');
         $table->float('price', 8, 2);
         $table->timestamps();
      });
   }

   public function down() {
      Schema::dropIfExists('widgets');
   }
}
```

The usefulness of the down method is frequently debated; some people advocate there is no practical reason you would ever roll *back* a migration in production but instead always roll forward to prevent data loss. Having worked on Laravel applications with sixty or more migration files spanning over a year, I can certainly agree you may never need all of those down statements. I always recommend getting comfortable with writing up and down methods. The down method will force you to think about what you're doing to the database in reverse and often times you may discover a bug or some other planned feature may have different needs.

To apply the changes in your migration run the command:

```
php artisan migrate
```

Now we are ready to move on to the model. A model is a class used to store and retrieve information about a particular database table. We can easily create a new model for our widgets table:

```
php artisan make:model Widget
```

Listing 4 shows our model stored in the `app/` folder.

We have added the protected array `fillable` to allow us to set those fields elsewhere in our application dynamically. If you do not set the fields and try to assign them you will get an error (empty data saved). Note: you do not have to specify the `timestamp` field in your model; the framework handles these fields for you automatically.

Now we have a migration which has created a table for us—a model that represents our table and will allow us to store and retrieve data—we can add a model factory. Model factories are somewhat new to the Laravel framework, and I find them extremely useful for creating sample data for your database based on your models which is even more useful for writing your tests. As you can guess, a model factory is a function you can call to create an instance of your model.

Model factories are defined in the `database/factories/ModelFactory.php` file. You can see an example of the `User` model here.

Listing 5 shows our model stored in the `database/factories/ModelFactory.php`.

We are leveraging the use of the package Faker[4] to create fake data for our widget. This callback will create a new instance of a widget and save it to

4 *Faker:* https://github.com/fzaninotto/Faker

Listing 4

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Widget extends Model
{
   protected $fillable = [
      'name',
      'description',
      'price',
   ];
}
```

Listing 5

```php
/** @var \Illuminate\Database\Eloquent\Factory $factory */
$factory->define(
   App\Widget::class,
   function (Faker\Generator $faker) {
      return [
         'name' => $faker->word . ' Widget',
         'description' => $faker->paragraph(3),
         'price' => $faker->randomFloat(2, 20, 999999),
      ];
   }
);
```

the database using fake data. Faker will create a random word, paragraph description, and price for our new widget. This allows us to create sample data easily.

The first place I normally utilize model factories is in my database seeder. I don't want to use production data during development, so I combine model seeders to create fake data which mimics production data and uses database seeders to create a realistic data set across all of my models. You can create individual data seeder class files, or you can use the existing `DatabaseSeeder.php` from the `database/seeds` folder. To create 20 widgets in our application, we can use a model factory in the database seeder file: `database/seeds/DatabaseSeeder.php` shown in Listing 6.

Listing 6

```php
<?php

use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder
{
   public function run() {
      factory(App\Widget::class, 20)->create();
   }
}
```

Similar to running our migration, we can seed our database by running an Artisan command:

```
php artisan db:seed
```

Now, if we inspect the database, we can see 20 rows of sample widget data. You can run database seeds as many times as you want; they will continue adding new rows each time.

If something happens to your database you can easily reset everything with `artisan`:

```
php artisan:reset
```

This will rollback all of the migrations, and you can run the migrate and seed commands again to set up your data.

From here, you're ready to go forth and start building fresh baked Artisanal applications with Laravel. You can easily set up routes, controllers, and start accessing data in a database. I look forward to seeing what you build.

Joe Ferguson is a PHP developer and community organizer. He is involved with many different technology related initiatives in Memphis including the Memphis PHP User group. He's been married to his extremely supportive and amazing wife for a really long time and she turned him into a crazy cat man. They live in the Memphis suburbs with their two cats. *@JoePFerguson*