**1. Introducción**

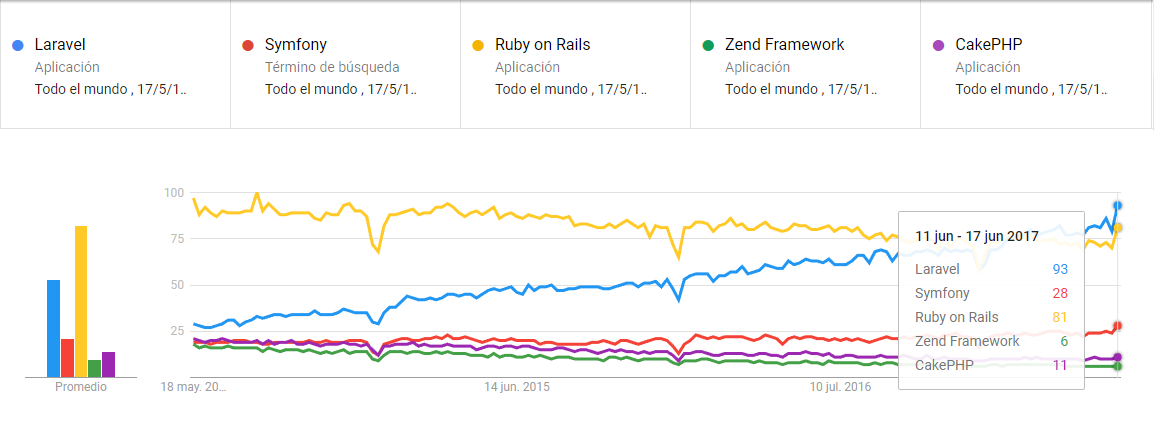
La curiosidad y ganas de aprendizaje sobre el mundo de la programación web me ha llevado a la creación de este trabajo.

El objetivo de este trabajo es la creación de un sistema gestor de contenidos “CMS“ utilizando el framework de PHP Laravel, el cual nos permitirá crear una estructura para la gestión y administración de contenidos de nuestra página web.

La idea principal de nuestro CMS será la de crear una interfaz que controlará una base de datos la cual almacenará el contenido de nuestra página web.

He escogido el framework Laravel, ya que es uno de los frameworks que más ha crecido en los últimos años.

Podemos observar el Interés a lo largo del tiempo de los principales frameworks de PHP y comprobamos que Laravel está en crecimiento, superando actualmente a Ruby on rails, el framework líder de la última década.



Comparativa de interès de los diferentes frameworks de PHP [Google Trends]

Que nos proporciona Laravel? [[1]](https://medium.com/@realvjy/why-laravel-is-best-php-framework-98a2784d76dc)

1. **Modularidad**: Posibilidad de separar los componentes. Laravel trabaja con modelo vista controlador [MVC].
2. **Motor de plantillas Blade**: Disponemos de un motor de plantillas con el cual podremos escribir código directamente sobre plantillas para ahorrarnos tiempo y esfuerzo.
3. **Auenticación**: Laravel mejora mucho en la simplicidad de la autenticación de usuarios, siendo esta parte imprescindible para el control de usuarios del gestor de contenidos.
4. **Almacenamiento en caché:** Laravel nos proporciona la posibilidad de gestionar la memoria caché directamente des de un fichero alojado en *config/cache.php*
5. **Laravel Inversion of Control (IoC)**: Herramienta para administrar dependencias de clase, donde una inyección de dependencias es un patrón de diseño orientado a objetos el cual subministra objetos a una clase en lugar de ser la propia clase la que cree el objeto.
6. **Enrutamiento**: Nos proporciona facilidad y un buen control en el enrutamiento.
7. **Controladores Restful**: Separación de la lógica para servir peticiones HTTP GET i POST.
8. **Generador de consultas de DB:** Laravel nos proporciona una interfaz para generar y ejecutar querys.

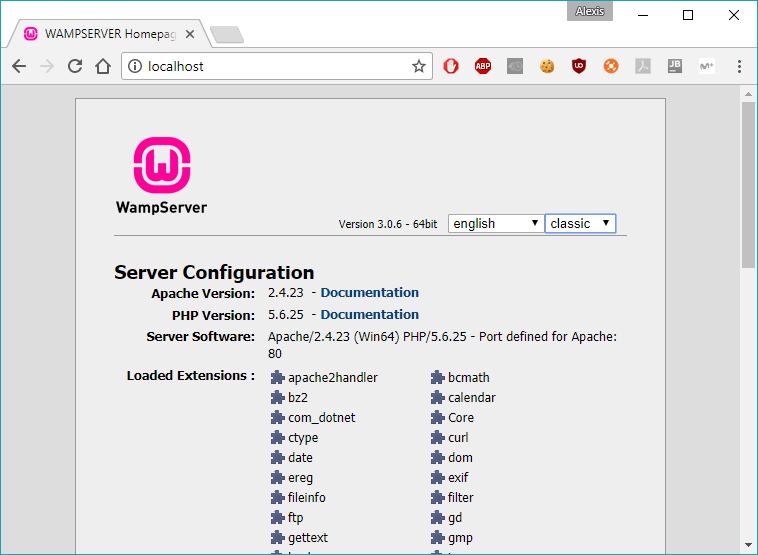
**2. Idea de la aplicación web**

**3. Entorno de trabajo**

1. **WampServer**

La aplicación correrá sobre Windows i se utilizará el entorno de desarrollo WampServer[[2]](http://www.wampserver.com/en/) que nos permite crear aplicaciones web con Apache2, PHP y MySQL.

Una vez instalado WampServer ya podremos acceder a nuestro navegador e ingresar la dirección localhost:



En Wamp la ruta donde añadimos nuestros proyectos es: wamp\www

1. **Composer**

Composer[[3]](https://getcomposer.org/) es el gestor de paquetes que nos proporcionará un formato estándar para gestionar las dependencias del software PHP y sus librerías, muy similar al “npm” de Node.js. Lo descargamos de la web y solo hay que añadirle la ruta del ejecutable del php.

Al finalizar la instalación, podremos comprobar la versión de composer de una manera sencilla:



1. **Laravel**

La actual versión de Laravel[[4]](https://laravel.com/docs/5.4) en la creación de este trabajo es la 5.4.

Laravel utiliza composer para gestionar sus dependencias, así que instalaremos Laravel utilizando Composer:

composer global require “laravel/installer”

Esta comanda nos instala laravel y nos genera los ficheros adecuados en el directorio

C:\Users\Alexis\AppData\Roaming\Composer

Entre los ficheros generados se encuentra el json adecuado con el siguiente contenido:

{

“require”:{ “laravel/installer”:”^1.3”}

}

Podemos comprobar nuestra versión de Laravel instalada.



1. **Crear proyecto**

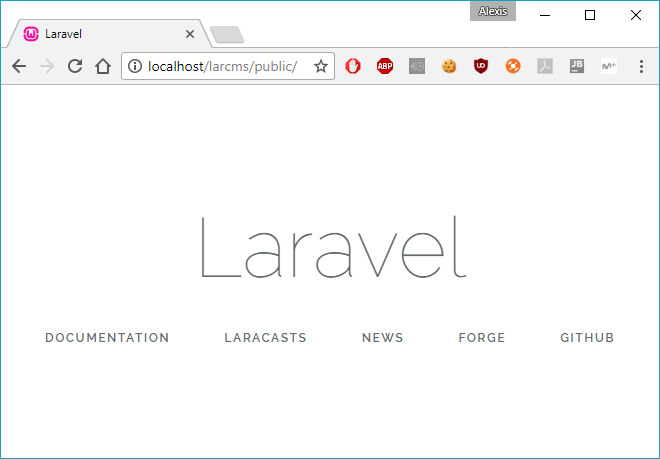
Crearemos un proyecto en la carpeta wamp\www de la siguiente manera:

laravel new larcms

Ejecución del proyecto

php artisan serve

A continuación ya podremos acceder des del navegador a la carpeta pública de nuestro proyecto: “localhost/larcms/public” o bien “localhost:8000”

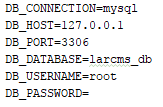


1. **Creación de la base de datos**

Utilizaremos MySQL[[5]](https://dev.mysql.com/downloads/mysql/) para gestionar la base de datos, con el usuario root y sin contraseña podremos acceder a la consola de MySQL.

CREATE DATABASE larcms\_db

Y en el fichero “.env” situado en la raíz de nuestro proyecto seleccionaremos la base de datos como la base de datos a utilizar.



**4. Elementos Utilizados de Laravel**

**4.1 Rutas**

Las rutas de Laravel las encontramos en routes/web.php i nos permiten realizar acciones según la url.

Un ejemplo de una ruta utilizada es el siguiente:

/\*  
| Ruta del índice del frontend, muestra la lista de posts.  
\*/  
Route::get('/', function () {  
 return view('frontend.index');  
});

La ruta anterior indicará que al acceder a la ruta ‘ / ’ la cual es la url básica de nuestra web, nos devolverá la vista index, situada en “view/frontend/index”.

**4.2 Database: Migrations**

Las migraciones en Laravel equivalen al control de versiones para la base de datos, pudiendo modificar el esquema de la base de datos de la aplicación, nos permitirá crear y manipular tablas en todos los sistemas de base de datos compatibles con Laravel.

Todas nuestras tablas seguirán el patrón “lc\_nombreTabla” siguiendo el formato de WordPress con sus tablas “wp\_...”.

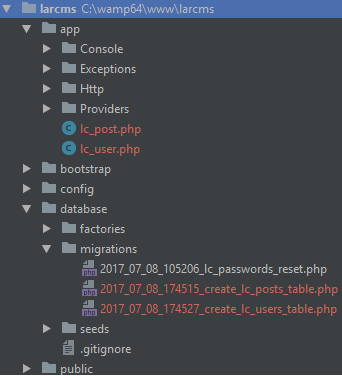
Mostraremos como creando un modelo y una migración podremos crear tablas en nuestra base de datos

Creación de la tabla que almacenará los posts de nuestro CMS.

Para crear una tabla, crearemos un modelo por comada que automáticamente creará el fichero migration. Dentro del fichero migration podremos crear los campos de nuestra tabla. La opción –m al crear un modelo, nos associará una migration.

Para las tablas que no necesiten modelo, podremos hacer “make:migration” en vez de make:model

php artisan make:model lc\_post -m



Podemos ver como nos ha creado los modelos “lc\_post y lc\_users” y también la comanda nos creará la migración identificada por la fecha de creación, y todas las migraciones quedaran almacenadas en la carpeta “migrations”.

Para añadir los campos a una tabla dentro de la migration lo haremos de la siguiente manera:

public function up()  
{  
 Schema::create('lc\_posts', function(Blueprint $table){  
 $table->increments('id');  
 $table->integer('autor\_id')->unsigned();  
 $table->foreign('autor\_id')->references('id')  
 ->on('lc\_users’)->onDelete('restrict');  
 $table->string('title');  
 $table->string('slug',250)->unique();  
 $table->text('excerpt');  
 $table->text('body');  
 $table->string('imagen')->nullable();  
 $table->timestamps();  
 });  
}

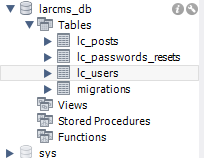
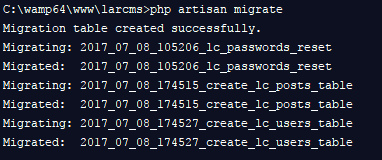
Un Post estará identificado por una ID i hará referencia a la id de un usuario como clave foránea. Tendrá un título, descripción e imagen de cabecera.

El término “Slug” hace referencia al título de una noticia, es un término muy importante para el posicionamiento SEO, cambia espacios por guiones y elimina caracteres especiales.

Por otro lado encontramos el campo excerpt, que de la misma manera que WordPress, permite a los usuarios mostrar más contenido en menos espacio, para mostrar contenido en la portada con enlaces a artículos completos.

Finalmente para realizar la migración, ejecutaremos el siguiente comando:

php artisan migrate

 Podremos comprobar el resultado creando usuarios, passwords\_resets y posts.

**Crear una migración para añadir un campo a la tabla lc\_posts.**

Si nos olvidamos de un campo de nuestra tabla, lo podemos añadir fácilmente. Como las migraciones consisten en un control de versiones, crearemos una nueva migración la cual asociaremos a la tabla ‘lc\_posts’ i automáticamente nos generará el schema necesario para modificar nuestra tabla. De la siguiente manera:

php artisan anadir\_fecha\_publicacion\_columna –table=lc\_posts

Y a continuación solo tendremos que añadir el campo fecha de publicación a nuestra tabla de posts.

public function up()  
{  
 Schema::table('lc\_posts', function (Blueprint $table) {  
 $table->timestamp('published\_at')->nullable();  
 });  
}

De la misma manera en la función down podemos hacer que elimine la tabla si queremos hacer un revert.

public function down()  
{  
 Schema::table('lc\_posts', function (Blueprint $table) {  
 $table->dropColumn('published\_at');  
 });  
}

Finalmente ya podemos ejecutar la migration.

php artisan migrate

**4.3 Database: Seeding**

Laravel incluye un método simple para inicializar los valores de una base de datos, pudiendo inicializar nuestras tablas.

Dentro del fichero DatabaseSeeder se hacen todas las llamadas a los seeders.

public function run()  
{  
 $this->call(UsersSeeder::*class*);  
 $this->call(PostsSeeder::*class*);  
}

Un seed será un fichero php que almacenarà los datos de inicialización de la tabla, un ejemplo es la creación de un usuario inicial:

public function run()  
{  
 // Reseteamos la tabla de usuarios  
 DB::table('lc\_users')->truncate();  
  
 // Generamos el usuario inicial  
 DB::table('lc\_users')->insert(  
 [  
 'name' => 'Alexis',  
 'email' => 'Alexis@udg.com',  
 'password' => bcrypt('secret')  
 ]  
 );  
}

Inicializamos tanto usuarios como posts utilizando seeds, nos ayudamos a la generación de post con la librería para crear frases y palabras aleatorias “Faker/Factory” ya incluida en el composer.

Cuando ya hemos creado los ficheros seeds, procedemos a su ejecución:

php artisan db:seed

Tabla Users;

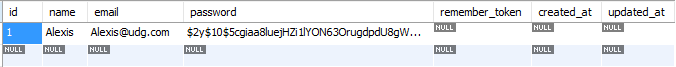
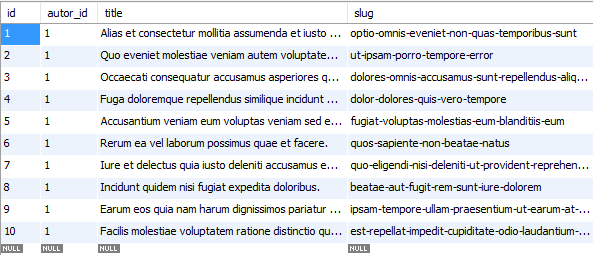


Tabla Posts:



**4.4 Controller**

**4.5 Eloquent: Accessors and mutators**

**4.5 Service Providers**

**4.6 Laravel Authentication**

Php artisan make:auth

Nos crea el controlador HomeContrller y En nuestras rutas automáticamente nos genera el código.

Auth::routes();  
  
Route::get('/home', 'HomeController@index')->name('home');

**4.7 Api’s utilizadas**

**Faker**

**Carbon:** Clase que hereda de DateTime de php

Gravatar: https://es.gravatar.com/

**Laravel Markdown** [**[x]**](https://github.com/GrahamCampbell/Laravel-Markdown)

El markdown es un lenguaje de marcado ligero con el que conseguiremos simplicidad para nuestras publicaciones.

Para el proyecto se ha utilizado Laravel Markdown, creado por Graham Cambell, su instalación es sencilla, para ello ayudándonos de composer, instalaremos el Markdown en nuestro proyecto de la siguiente manera:

Composer require Graham-campbell/markdown

Una vez instlado tenemos que registrar el provedor de servicio en nuestro fichero config/app.php dentro de la lista de providers:

GrahamCampbell\Markdown\MarkdownServiceProvider::*class*,

De la misma manera añadimos el alias a la lista de alias.

'Markdown' => GrahamCampbell\Markdown\Facades\Markdown::*class*,

El uso de markdowns es simple, la base de datos almacena el código en formato markdown, y el php se encarga de convertirlo a HTML, en la siguiente imagen podemos ver el efecto.



Para una mayor limpieza del código http, le encargamos la funcionalidad de convertir a html al modelo. No sin antes importar los datos en el modelo.

use GrahamCampbell\Markdown\Facades\Markdown;

Modelo “lc\_post”

public function getBodyHtmlAttribute($value) {  
 return Markdown::convertToHtml(e($this->body));  
}

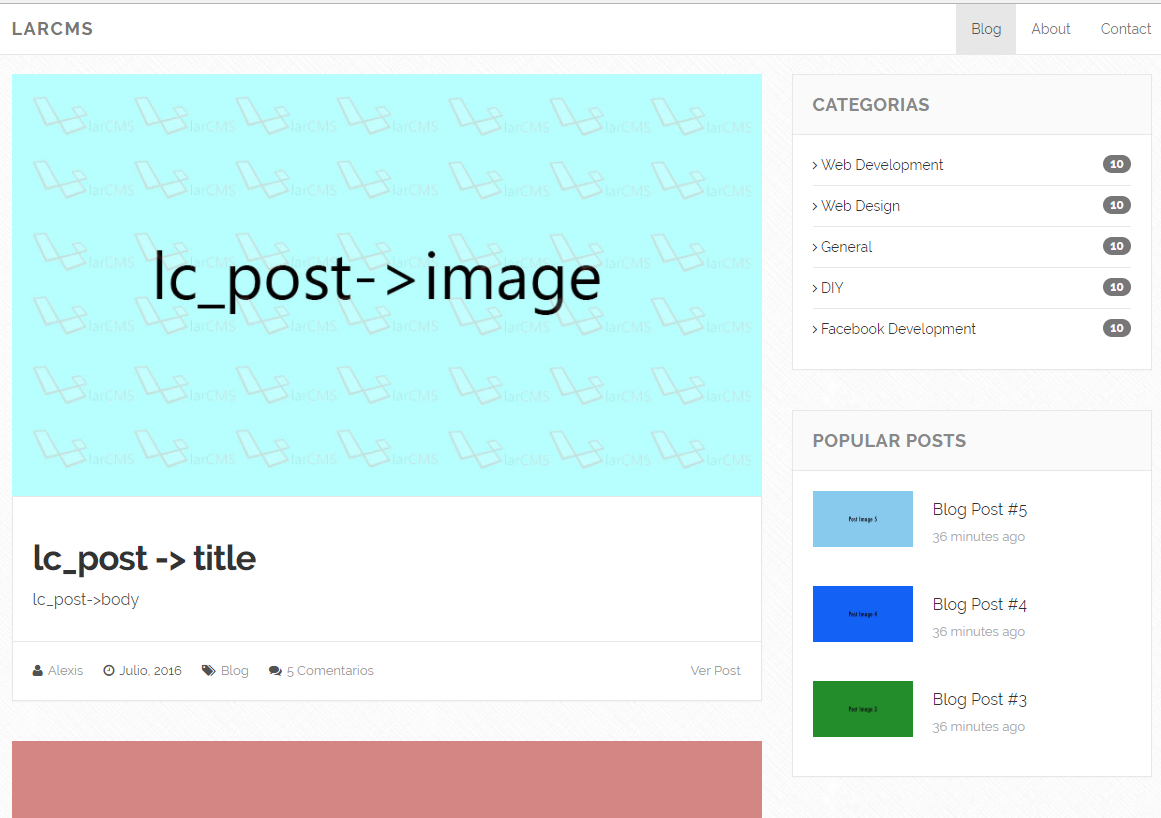
Vista “post”

<div>  
 {!! $post->body\_html !!}  
</div>

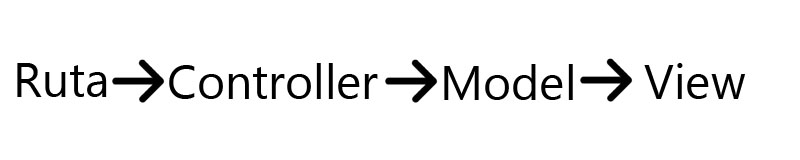
**5. Frontend**

**5.1 Mostrando el blog**

En este paso observaremos como recuperar información de la base de datos, concretamente de la tabla lc\_posts para mostrarla dinámicamente en la lista principal de posts. Podemos ver el formato de un post en la lista de la web.



Uso de controllers



Para mostrar la lista de posts necesitaremos crear un controller. Los controller pueden organizar toda la lógica de gestión de solicitudes en una sola clase situada en app/http/controllers.

Si queremos mostrar la lista de posts a la que llamaremos “blog”, crearemos un controller para gestionar su lógica de gestión de solicitudes.

php artisan make:controller BlogController

Para que una ruta tome como acción un controller lo tenemos que indicar en el fichero de rutas.

/\*  
| Ruta del índice del frontend, muestra la lista de posts.  
\*/  
Route::get('/', [  
 'uses' => 'BlogController@index',  
]);

En la ruta indicamos que va a utilizar el controlador BlogController y la función index. En el controller recogeremos los datos del modelo para enviarlos a la vista. De la siguiente manera:

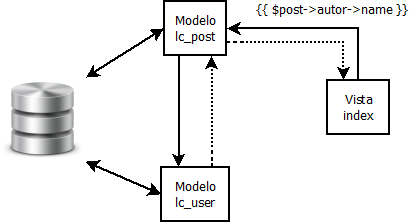
class BlogController extends Controller  
{  
 public function index()  
 {  
 $posts = lc\_post::all();  
  
 return view("frontend.index", compact('posts'));  
 }  
}

y a continuación ya podemos recoger los datos en la vista, almacenados en la variable @posts.

**5.2 Recuperar datos del usuario/categoria en un post**

Recuperar Datos del usuario de un post

Para recuperar los datos de un usuario, la vista que muestra la lista de post le pregunta al modelo que tiene el post actual: ¿cuál es el usuario que ha redactado el post? El modelo que guarda el post solo conoce la id del usuario, así que le envía la pregunta al modelo usuario.



**5.2 Formato de una publicación**

**5.3 SEO-Friendly**

Para hacer que las publicaciones sean encontradas fácilmente des de nuestros navegadores, tendremos que cumplir unas características que abarquen diferentes aspectos.

* Contenido optimizado: Las imágenes llevaran por etiqueta “ALT” el nombre del título, por otra parte, no repetiremos uso de las etiquetas “Heading” marcando “H1” una única vez como elemento principal y destacado.
* Metas:
* Diseño web: El diseño no solo debe ser atractivo de cara al usuario, si no que también debe favorecer la navegación y accesibilidad, adecuando el diseño responsive.
* Indexación: No utilizar contenidos Flash ya que no son indexables para los buscadores.
* Url-amigables: Nuestras urls deben ser simples y efectivas. Indexar una publicación por ID no es efectivo, así que cuando generemos una publicación podremos escoger el “SLUG” donde escogeremos el formato de nuestra URL.

El formato de la url de una publicación es el siguiente:

http://localhost:8000/blog/slug-de-la-noticia

Para modificar el valor de nuestra URL tenemos que modificar el proveedor que nos sirve rutas, se encuentra en la clase RouteServiceProvider.

Para personalizar la lógica de resolución de rutas usamos el método Route::bind, el cual recibe el valor del segmento de la URL y devuelve la instancia de la clase que debe ser inyectada en la ruta.

Por lo tanto si queremos personalizar la URL de un post y que sea accedible utilizando el SLUG, necesitamos llamar al modelo del post.

public function boot()  
{  
 parent::boot();  
  
 Route::bind('post',function($slug){  
 return lc\_post::publicado()->where('slug',$slug)->first();  
 });  
}

**5.4 Filtrando por categoría**

**5.5 Filtrando por autor**

**5.6 Página de error**

**6. Backend**

**6.1 Estructura del backend**

Nuevo controller Backend Controller.php

LoginController

**6.2 Mostrando la lista de publicaciones**

**6.3 Añadiendo una publicación**

**laravelCollective/html (video 31)**

[**https://laravelcollective.com/docs/5.2/html**](https://laravelcollective.com/docs/5.2/html)

**Mirar requests**

**Creación de thumbnails** [**https://packagist.org/packages/intervention/image**](https://packagist.org/packages/intervention/image)

**Simplemde** [**https://simplemde.com/**](https://simplemde.com/)

**Bootstrap datetimepicker** [**https://eonasdan.github.io/bootstrap-datetimepicker/**](https://eonasdan.github.io/bootstrap-datetimepicker/)

**Jasny Bootstrap** [**http://www.jasny.net/bootstrap/**](http://www.jasny.net/bootstrap/)

¡

**7. Bibliografia y enlaces web**

Comparativa: <https://medium.com/@realvjy/why-laravel-is-best-php-framework-98a2784d76dc> [1]

WampServer: <http://www.wampserver.com/en/> [2]

Composer: <https://getcomposer.org/> [3]

Laravel: <https://laravel.com/docs/5.4> [4]