Índice

Lavarel

Laravel es un popular framework de PHP. Permite el desarrollo de aplicaciones web totalmente personalizadas de elevada calidad.

Laravel es un framework PHP. Es uno de los frameworks más utilizados y de mayor comunidad en el mundo de Internet.

Como framework resulta bastante moderno y ofrece muchas utilidades potentes a los desarrolladores, que permiten agilizar el desarrollo de las aplicaciones web.

Laravel pone énfasis en la calidad del código, la facilidad de mantenimiento y escalabilidad, lo que permite realizar proyectos desde pequeños a grandes o muy grandes. Además permite y facilita el trabajo en equipo y promueve las mejores prácticas.

Las características más notables que aporta Laravel son las siguientes:

- Blade: Blade es un sistema de plantillas para crear vistas en Laravel. Este permite extender plantillas creadas y secciones en otras vistas en las cuales también tendremos accesibles las variables y con posibilidad de utilizar código PHP en ellas
- Eloquent: Eloquent es el ORM que incluye Laravel para manejar de una forma fácil y sencilla los procesos correspondientes al manejo de bases de datos en nuestro proyecto. Transforma las consultas SQL a un sistema MVC lo que no permite procesar consultas SQL directamente y así protegernos de la inyección SQL.
- Routing: Laravel proporciona un sistema de organización y gestión de rutas que nos permite controlar de manera exhaustiva las rutas de nuestro sistema.
- Middlewares: Son una especie de controladores que se ejecutan antes y después de una petición al servidor, lo que nos permite insertar múltiples controles, validaciones o procesos en estos puntos del flujo de la aplicación.
- Comunidad y documentación: Un gran punto a destacar de este framework es la gran comunidad y documentación que existe, una comunidad de profesionales activa que aporta conocimiento y funcionalidades, además de testear nuevas versiones y detectar fallos del framework, lo que le da seguridad al framework. Y una documentación muy completa y de calidad pensada para los propios desarrolladores.

La página oficial de Laravel es https://laravel.com

Actualmente Laravel está en su versión 8 pero vamos a usar la última versión LTS (Long Term Support) que es la versión 6.

MVC

El MVC (modelo, vista, controlador) es un patrón arquitectónico de software que separa una aplicación en tres capas descritas como su acrónimo lo indica. Laravel, así como la mayoría de frameworks en PHP implementan este patrón de diseño en donde cada capa maneja un aspecto de la aplicación. Pero antes de ver cómo Laravel está diseñado para implementar

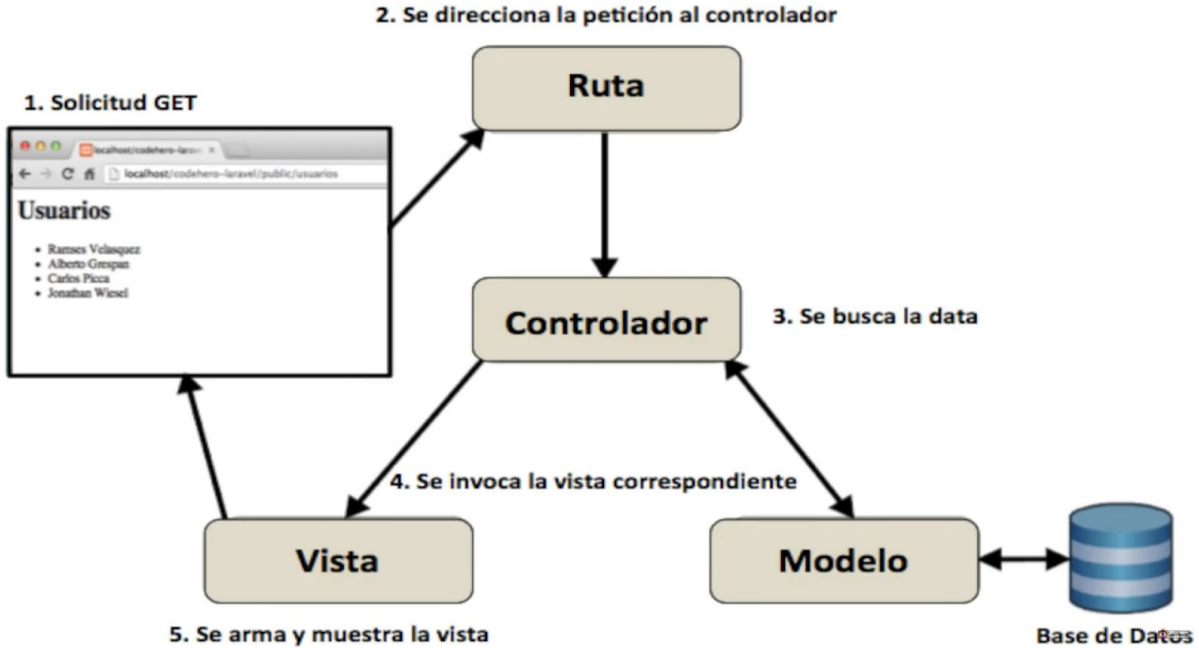
este patrón de software, vamos a tratar de dejar este concepto un poco más claro definiendo cada una de sus partes.

Modelo: Hace referencia a la estructura de datos de la aplicación. Los datos pueden ser transferidos desde la base de datos, una clase, un servicio, u otros, directamente a la vista o ser transformados en el controlador para ser actualizados nuevamente al origen.

Vista: Es la representación de la información en una interfaz de usuario. Por lo general en interfaces no estáticas se representan los datos que vienen directamente del modelo o estos son transformados en un proceso intermedio en el controlador. En vistas estáticas por lo general no hace falta que las vistas sean renderizadas con datos enviados del controlador.

Controlador: Es el lugar en donde se implementa la lógica de la aplicación, los procedimientos, algoritmos y rutinas que hacen que funcione el software. Actúa como interfaz entre los componentes de modelo y vista aplicando las transformaciones y lógica necesarias.

¿Cómo funciona Laravel?



1. Se hace una petición por medio de una ruta.
2. La ruta llama a un controlador.
3. El controlador se encarga de extraer los datos de la base de datos con ayuda del modelo.
4. El controlador mando los datos extraídos a la vista.
5. La vista retorna los datos obtenidos.

Pre-requisitos para trabajar con Laravel

En este curso trabajaremos con Laravel 6, el cual requiere de PHP 7.2 o superior, la extensión PDO para el trabajo con base de datos, así como de otras extensiones que puedes ver en la documentación de Laravel.

Requisitos que debe cumplir el servidor:

- PHP >= 7.2.0
- Extensión BCMath para PHP
- Extensión Ctype para PHP
- Extensión Fileinfo para PHP
- Extensión JSON para PHP
- Extensión Mbstring para PHP
- Extensión OpenSSL para PHP
- Extensión PDO para PHP
- Extensión Tokenizer para PHP
- Extensión XML para PHP

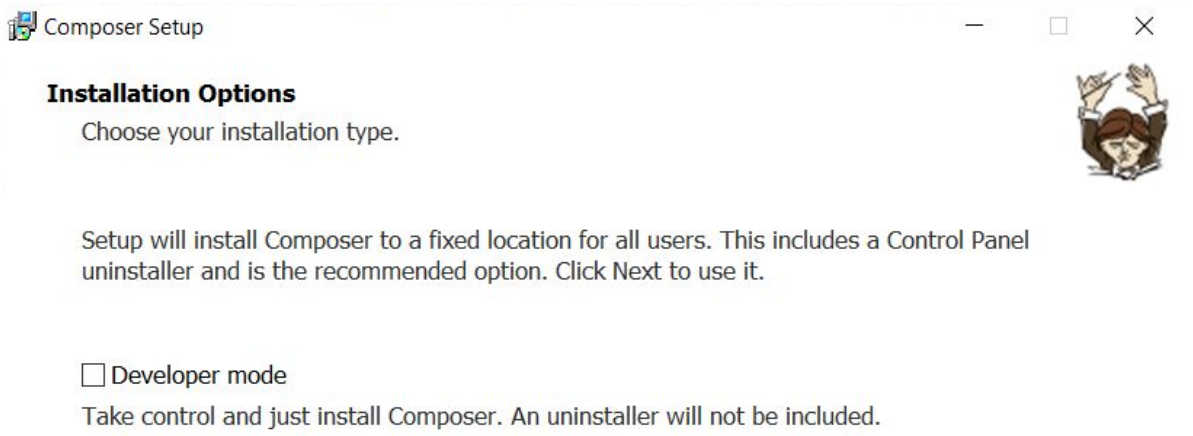
Al instalar XAMPP ya tenemos todas las extensiones necesarias para trabajar con Laravel con lo que sólo necesitaremos instalar “Composer”.

Instalación de Composer

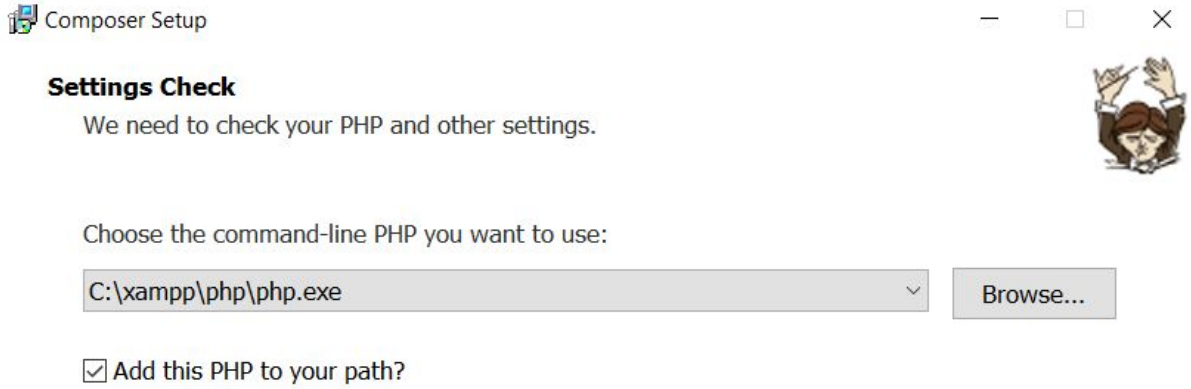
Antes de instalar frameworks y componentes de PHP debemos instalar Composer. Composer es el manejador de dependencias para PHP. Una dependencia puede ser tanto un framework (como Laravel o Symfony) así como paquetes o componentes.

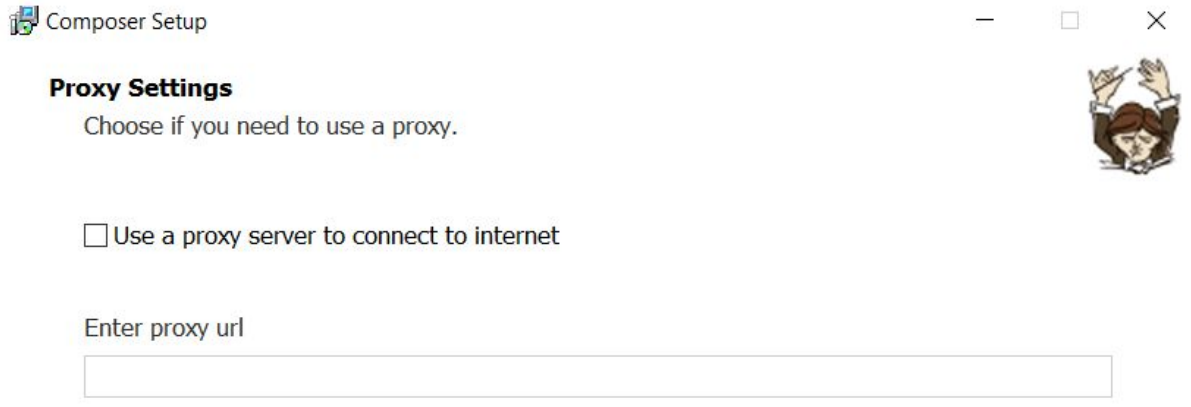
Composer nos permitirá instalar y actualizar el framework y todos los componentes de PHP que requiramos en nuestro proyecto con tan solo ejecutar un comando, es decir, sin tener que buscar y descargar archivos, descomprimirlos, copiarlos, pegarlos, etc. Esto es muy conveniente sobretodo para mantener nuestro proyecto actualizado con correcciones de bugs y parches de seguridad. Composer también nos permitirá detectar problemas entre nuestras dependencias, por ejemplo si una versión de un componente no es compatible con nuestra versión de PHP o del framework Laravel, etc.

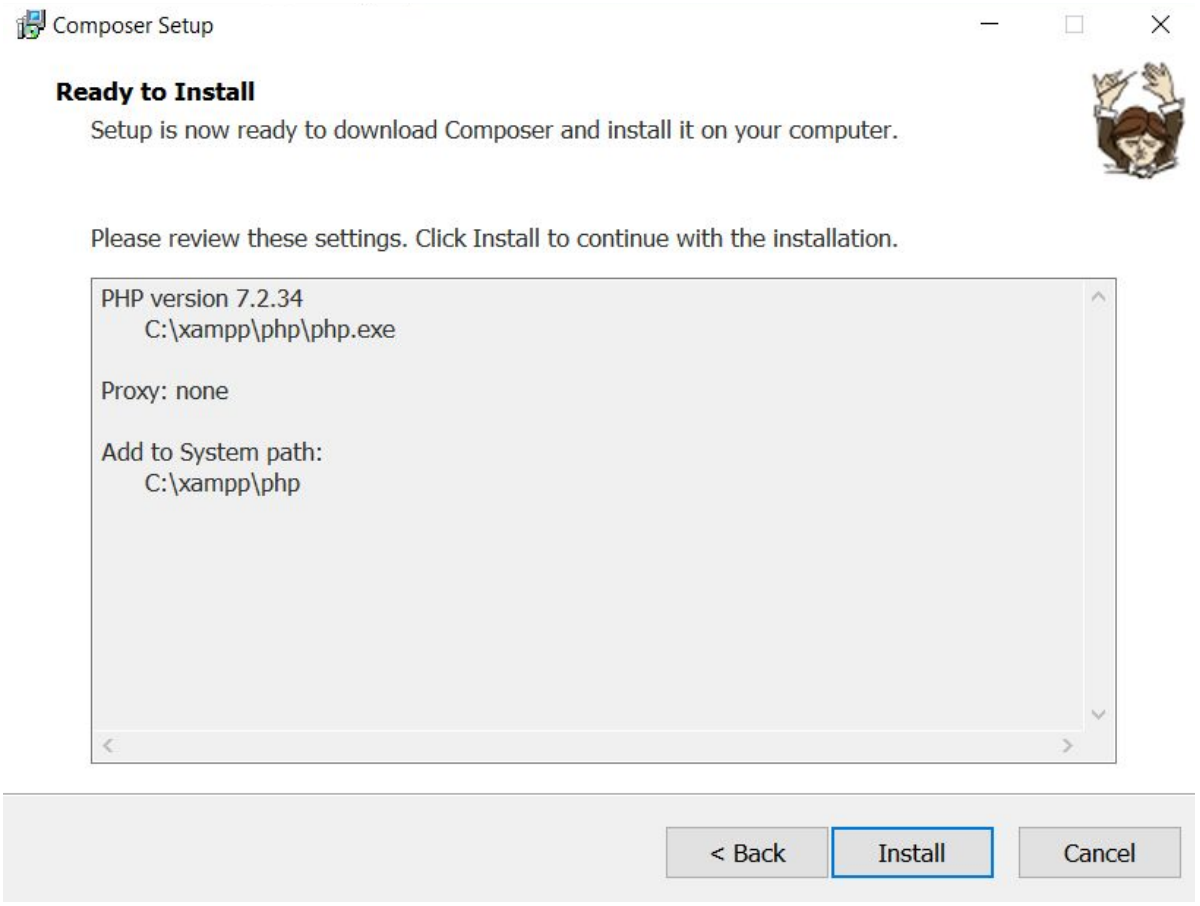
Instalar Composer en Windows es muy sencillo gracias a su instalador que puedes descargar desde aquí: https://getcomposer.org/Composer-Setup.exe. Este instalador instalará la última versión de Composer y configurará el PATH por ti para que puedas ejecutar composer en tu consola desde cualquier directorio.

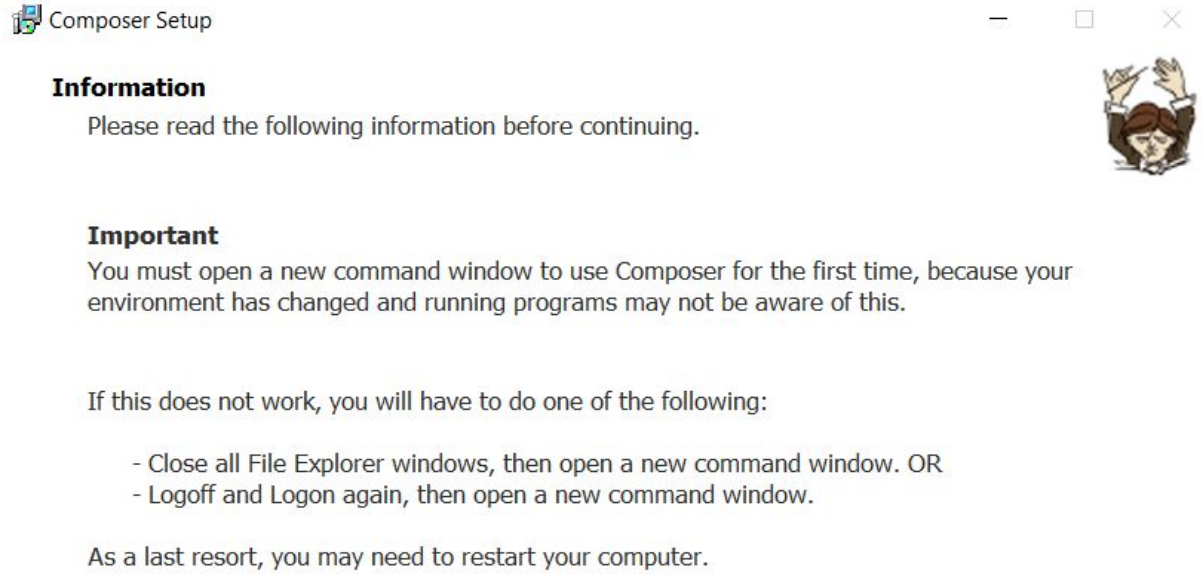


En la instalación “Developer Mode” significa que seleccionará automáticamente la ruta de PHP si tienes más de una versión de PHP. Si se prefiere una versión específica, tiene que estar desmarcada la opción.









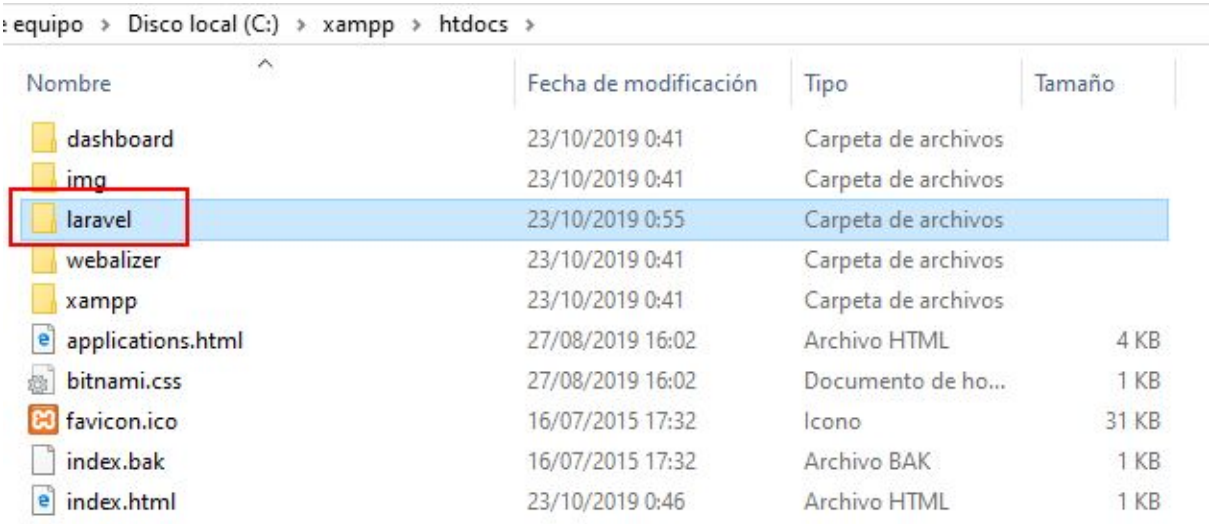
Después de completar la instalación, abre el símbolo del sistema. Escribe «cmd» y haz clic en OK. A continuación Ingresa el siguiente comando:

```
composer
```

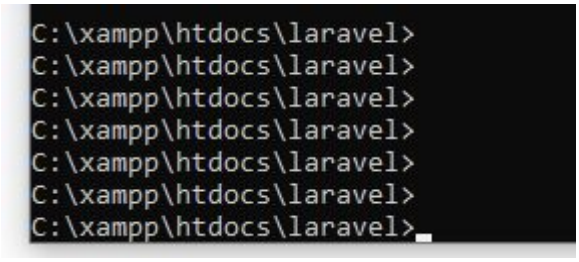
Instalar Laravel

Ahora ya estamos listos para instalar Laravel.

Nos dirigiremos a nuestro directorio htdocs de nuestro XAMPP y crearemos una carpeta llamada «laravel».



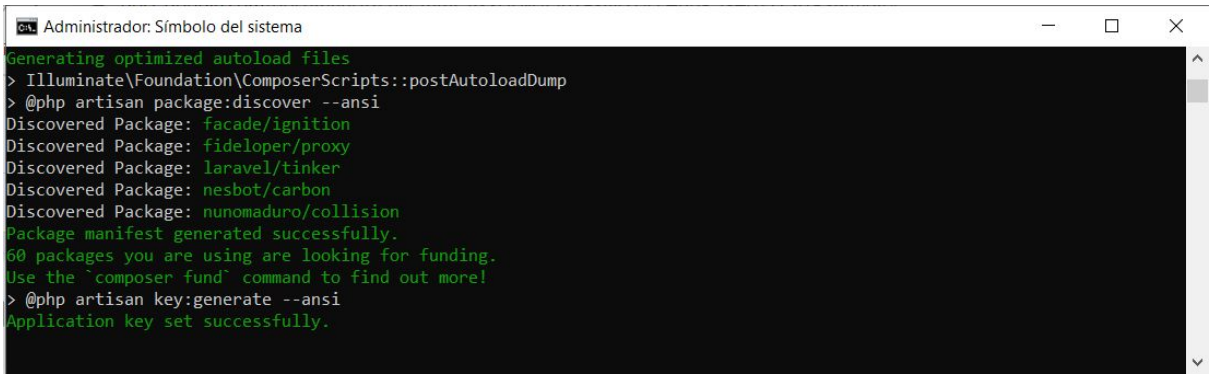
Una vez creada, desde la consola de MSDOS nos dirigiremos a la carpeta que hemos creado.



Desde ahí ejecutaremos el siguiente comando, donde el último «laravel» es el proyecto que vamos a crear.

```
composer create-project --prefer-dist laravel/laravel laravel "6.*"
```

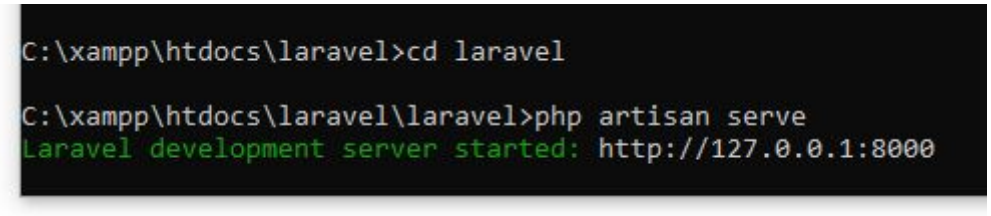
Este comando iniciará la instalación del Framework. Tardará unos cuantos minutos dependiendo de la conexión a Internet que tengamos. Si todo termina correctamente nos mostrará algo similar al siguiente pantallazo.



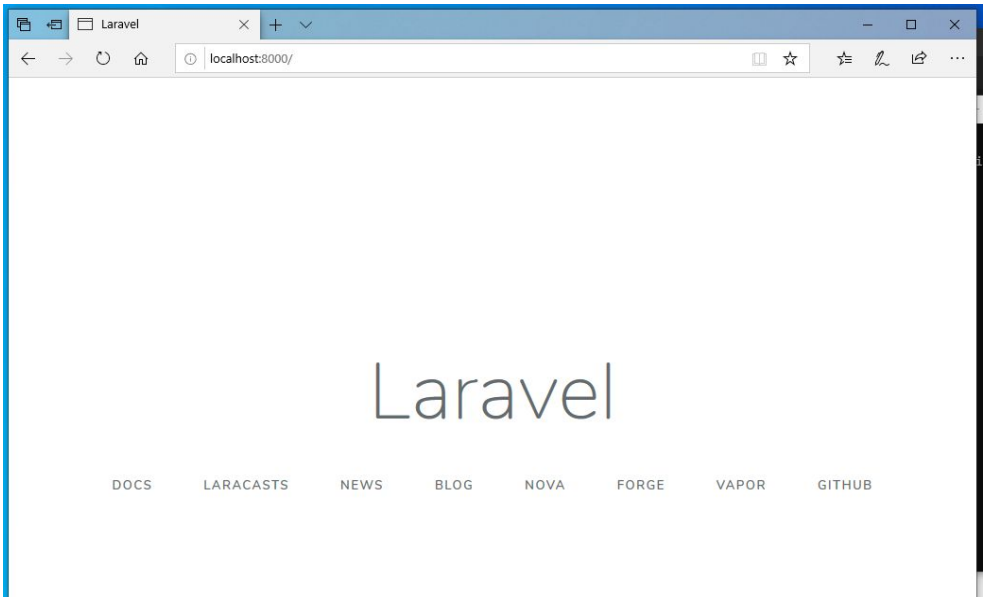
Ahora, para iniciar el servicio de Laravel, desde la ventana de MSDOS ejecutaremos el siguiente comando.

```
php artisan serve
```

Y si el servicio inicia correctamente mostrará un mensaje como el de la captura siguiente con el servicio iniciado en el puerto por defecto 8000.



Ahora solo deberemos ir a nuestro navegador para comprobar que carga correctamente.



Otras opciones de desarrollo local más robustas están disponibles mediante Homestead y Valet.

Estructura de directorios

La estructura por defecto de aplicación de Laravel está pensada para proporcionar un buen punto de inicio para aplicaciones grandes y pequeñas. Pero, eres libre de organizar tu aplicación como quieras. Laravel no impone casi ninguna restricción sobre dónde una clase es ubicada – siempre y cuando Composer pueda cargar automáticamente la clase.

Directorio App

El directorio app contiene el código principal de tu aplicación. Casi todas las clases en tu aplicación estarán en este directorio.

Directorio Bootstrap

El directorio bootstrap contiene el archivo app.php que maqueta el framework. Este directorio también almacena un directorio cache que contiene archivos generados por el framework para optimización de rendimiento como los archivos de caché de rutas y servicios.

Directorio Config

El directorio config, como el nombre implica, contiene todos los archivos de configuración de tu aplicación. Es una buena idea leer todos estos archivos y familiarizarte con todas las opciones disponibles para ti.

Directorio Database

El directorio database contiene las migraciones de tu base de datos. Si lo deseas, puedes también usar este directorio para almacenar una base de datos SQLite.

Directorio Public

El directorio public contiene el archivo index.php, el cual es el punto de acceso para todas las solicitudes que llegan a tu aplicación y configura la autocarga. Este directorio también almacena tus assets, tales como imágenes, JavaScript y CSS.

Directorio Resources

El directorio resources contiene tus vistas así como también tus assets sin compilar tales como LESS, Sass o JavaScript. Este directorio también almacena todos tus archivos de idiomas.

Directorio Routes

El directorio routes contiene todas las definiciones de rutas para tu aplicación. Por defecto, algunos archivos de rutas son incluidos con Laravel: web.php, api.php, console.php y channels.php.

Directorio Storage

El directorio storage contiene tus plantillas compiladas de Blade, sesiones basadas en archivos, archivos de caché y otros archivos generados por el framework. Este directorio está segregado en los directorios app, framework y logs. El directorio app puede ser usado para almacenar cualquier archivo generado por tu aplicación. El directorio framework es usado para almacenar archivos generados por el framework y caché. Finalmente, el directorio logs contiene los archivos de registros de tu aplicación.

El Directorio Tests

El directorio tests contiene tus pruebas automatizadas. Una prueba de ejemplo de PHPUnit es proporcionada. Cada clase de prueba debe tener el sufijo Test. Puedes ejecutar tus pruebas usando los comandos phpunit o php vendor/bin/phpunit.

Directorio Vendor

El directorio vendor contiene tus dependencias de Composer.

Configuración

Base de datos

Laravel hace que la interacción con las bases de datos sea extremadamente fácil a través de una variedad de backends de bases de datos usando SQL nativo, el constructor de consultas query builder y el ORM Eloquent. Actualmente, Laravel soporta cuatro bases de datos:

- MySQL 5.6+
- PostgreSQL 9.4+
- SQLite 3.8.8+
- SQL Server 2017+

La configuración de base de datos para tu aplicación está localizada en **config/database.php**. En este archivo puedes definir todas tus conexiones de bases de datos, y también especificar qué conexión debería ser usada por defecto. Ejemplos para la mayoría de los sistemas de bases de datos soportados son proporcionados en este archivo.

De todas las configuraciones definidas, ¿cuál es la que está usando Laravel? Si miramos sobre la línea 16 de este fichero, vemos lo siguiente:

```
'default' => env('DB_CONNECTION', 'mysql'),
```

La función env, nos está diciendo que usa la conexión **mysql**.

Antes de empezar a trabajar en un proyecto tendremos que crear la base de datos. En nuestro caso el nombre que vamos a asignar a nuestra base de datos es **dwes_laravel**.

Archivo .env

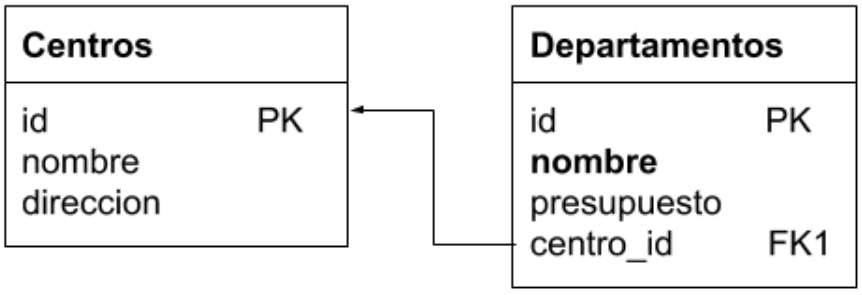
Una vez creada la base de datos abriremos el archivo **.env** que se encuentra en la raíz del proyecto. En este archivo tendremos que poner los datos de conexión de nuestra base de datos.

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=dwes_laravel
DB_USERNAME=root
DB_PASSWORD=
```

Toda la documentación del manejo de base de datos con Laravel la podemos encontrar en la siguiente página:

Base de datos en Laravel

Nuestro proyecto va a estar formado por dos tablas: Centros y Departamentos. Los campos y la relación entre las tablas es la siguiente:



Creación de tablas con el sistema de migraciones

Laravel incluye un sistema de migraciones de base de datos con el cual podemos definir todas las tablas de nuestra aplicación desde PHP, utilizando una interfaz orientada a objetos. Este sistema nos da la ventaja de que podemos guardar las diferentes versiones de nuestra base de datos como código dentro del sistema de control de versiones (por ejemplo git), además nos permite generar tablas para diferentes bases de datos (MySQL, PostgreSQL, SQLite, y SQL Server) usando el mismo código PHP.

Las migraciones de Laravel no son más que archivos de PHP ubicados en el directorio **database/migrations**. Una aplicación de Laravel viene con 3 migraciones por defecto, las cuales puedes modificar o eliminar. Se llaman migraciones ya que heredan de la clase migration.

La primera migración trae lo necesario para crear la tabla donde se guardará la información de cada usuario incluyendo su email y contraseña. La segunda es la migración que utiliza Laravel para guardar la información para la recuperación de contraseñas y la tercera es un poco más avanzada y no vamos a verla en este curso.

Así que los pasos para crear una tabla dentro de una base de datos en Laravel son:

1. Crear fichero migration.
2. Configurar archivo migration.
3. Ejecutar la migración (comando php artisan make:migration) que crea la tabla en la base de datos.

Si abrimos el fichero de migration CreateUsersTable veremos lo siguiente:.

```
<?php

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

class CreateUsersTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('users', function (Blueprint $table) {
```

```
            $table->bigIncrements('id');
            $table->string('name');
            $table->string('email')->unique();
            $table->timestamp('email_verified_at')->nullable();
            $table->string('password');
            $table->rememberToken();
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('users');
    }
}
```

Si nos fijamos en este archivo, veremos que no es más que una clase PHP que hereda de la clase Migration con dos métodos. El método up que es el que crea la tabla y el método down que elimina la tabla.
Dentro del método up, ofrece una clase **Schema** con un método estático **create,** que es el que hace toda la tarea. Este método tiene dos argumentos uno el nombre de la tabla, y el segundo una función anónima que tiene un parámetro table que utiliza para confeccionar los campos de la tabla.

Si queremos crear estas tablas en nuestra base de datos sólo tendremos que ejecutar

```
php artisan migrate
```

Si recibes un error y no se ha creado ninguna tabla, es posible que no hayas configurado tu base de datos correctamente. En este caso regresa al archivo .**env** y arregla los valores.

Si el error ocurrió luego después de que se creara al menos una de las tablas, es posible que hayas definido una migración incorrectamente. Arregla el código y luego vuelve a ejecutar el comando **php artisan migrate**.

Alternativamente puedes ejecutar **php artisan migrate:fresh** para borrar todas las tablas y ejecutar las migraciones de nuevo y desde cero.

Debes tener cuidado con comandos como **php artisan migrate:fresh**, que ejecutan acciones destructivas como eliminar todas las tablas de tu base de datos. Este comando en particular es bastante útil en desarrollo pero sería desastroso ejecutarlo en un servidor de producción.

También puedes ejecutar el comando **php artisan migrate:rollback** para revertir las migraciones. Este comando ejecutará todos los métodos down de cada migración en el orden inverso en el que fueron ejecutadas previamente. En nuestro ejemplo deberás ver cómo se eliminan las tablas creadas previamente; a excepción de la tabla migrations que Laravel utiliza para llevar el control de las migraciones que ya han sido ejecutadas y el orden en que fueron ejecutadas.

Si deseas usar el comando **migrate:rollback**, asegúrate de agregar la lógica para revertir la migración en el método down().

Para crear nuestra propia migración desde la consola debemos usar el comando **php artisan make:migration** seguido del nombre de la migración, por ejemplo:

```
php artisan make:migration nombre_migracion
```

El nombre de la migración sigue una convención, así si queremos crear la tabla, ***centros*** el nombre de la migración a ejecutar debería ser:

```
php artisan make:migration create_centros_table
--create="centros"
```

El nombre create_create_table es una convención: create_[NOMBRE DE LA TABLA AQUÍ]_table.

Si abrimos el fichero de migración de la tabla Centro tendrá el siguiente aspecto.

```
<?php

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

class CreateCentrosTable extends Migration
{
    /**
     * Run the migrations.
     *
```

```
     * @return void
     */
    public function up()
    {
        Schema::create('centros', function (Blueprint $table) {
            $table->bigIncrements('id');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('centros');
    }
}
```

En este fichero hay dos métodos: up y down:

En el método up definiremos la acción principal de nuestra migración, que en este caso es la creación de la tabla **centros**. En el método down definiremos la acción contraria, en este caso la eliminación de la tabla de **centros**. Este método es opcional.

El método up trae por defecto la creación de una serie de campos:
$table->bigIncrements('id'); define una columna de tipo BIGINT UNSIGNED, es decir, entero grande sin números negativos y que se autoincrementa.
$table->timestamps(); define 2 columnas de tipo TIMESTAMP con los nombres created_at y updated_at, estas columnas son usadas por Eloquent, el ORM de Laravel que veremos más adelante.

Vamos a agregar 2 columnas más en el medio:

$table->string('nombre',100;; define una columna **nombre** de tipo VARCHAR y de tamaño 100.
$table->string(direccion); define una columna **direccion** de tipo VARCHAR

Con estos cambios el fichero de migración de la tabla Centros queda así:

```
<?php
```

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

class CreateCentrosTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('centros', function (Blueprint $table) {
            $table->bigIncrements('id');
            $table->string('nombre',100);
            $table->string('direccion');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('centros');
    }
}
```

Para la creación de la tabla departamentos tendremos que crear primero la migración con la siguiente instrucción:

```
php artisan make:migration create_departamentos_table --create="departamentos"
```

Al fichero creado le tendremos que añadir 3 columnas más en el medio:

$table->string('nombre',100->unique(); define una columna **nombre** única de tipo VARCHAR que de tamaño 100.
$table->unsignedIntege('presupuesto'); define una columna **presupuesto** de tipo INTEGER sin signo.
$table->unsignedBigInteger('centro_id'); define una columna **centro_id** de tipo BIGINT UNSIGNED sin signo para almacenar el valor del centro.

También tendremos que añadir la relación entre las columnas centro_id de la tabla departamentos y la columna id de la tabla centros. Esto se hace con la siguiente línea.

$table->foreign('centro_id')->references('id')->on('centros');

Con estos cambios el fichero de migración de la tabla Departamentos queda así:

```
<?php

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

class CreateDepartamentosTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
                Schema::create('departamentos', function (Blueprint
$table) {
            $table->bigIncrements('id');
            $table->string('nombre',100)->unique();
            $table->unsignedInteger('presupuesto');
            $table->unsignedBigInteger('centro_id');
            $table->timestamps();

$table->foreign('centro_id')->references('id')->on('centros');
        });
    }
```

```
    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('departamentos');
    }
}
```

Tras crear los ficheros de migraciones con la estructura de las tablas hay que trasladarlas a la base de datos. Para ello sólo tendremos que ejecutar el comando

```
php artisan migrate
```

Toda la documentación acerca de las migraciones de Laravel lo podemos encontrar en la siguiente página:

Migraciones en Laravel

Rutas

La rutas de nuestro proyecto se encuentran en el archivo **web.php** dentro de la carpeta **routes**.

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

En el código anterior nos encontramos con la clase **Route** y el método estático **get**. Dentro del método tenemos dos parámetros la URL donde quieres "/" ir y una función anónima que dice donde nos tenemos que dirigir al introducir la URL del primer parámetro.

Laravel crea por defecto la ruta de la página principal y retorna la vista **welcome** que se encuentra en carpeta **views** dentro de **resources**.

Para crear nuevas rutas basta con copiar el código anterior y modificar la ruta en el primer parámetro del método get y en la función anónima en lugar de devolver una vista vamos a devolver una cadena de texto.

```
Route::get('/ejemplo', function () {
    return "Página de ejemplo";
});
```

Nuestra rutas también pueden tener parámetros. Si queremos añadir un parámetro a nuestra ruta lo encerramos entre llaves. El nombre que pongamos es un identificador para ese parámetro. A continuación en los argumentos de nuestra función anónima tenemos que relacionar ese parámetro con una variable de PHP.

```
Route::get('/usuario/{id}', function ($id) {
    return "Bienvenido usuario:" .$id;
});
```

Si queremos pasar más de un parámetro a nuestra URL, basta con añadir otro nombre identificativo para ese segundo parámetro

```
Route::get('/usuario/{id}/{nombre}', function ($id, $name) {
    return "Bienvenido " . $name . " :" .$id;
});
```

En el código anterior el segundo parámetro lo suyo es que sea una cadena, pero tal y como está definido actualmente podría numérico o cualquier otra cosa. Si queremos filtrar ese segundo parámetro tenemos que recurrir a las expresiones regulares de PHP.

Para el ejemplo anterior si queremos que el nombre esté compuesto por caracteres entre la **a** y la **z** debemos añadir antes de finalizar la instrucción de Route un método “where” que el primer parámetro recibe el nombre del parámetro que queremos limitar y el segundo parámetro la expresión regular.

```
Route::get('/usuario/{id}/{nombre}', function ($id, $name) {
    return "Bienvenido " . $name . " :" .$id;
})->where('nombre','[a-zA-Z]+');
```

También es posible que queramos establecer un parámetro como opcional, en este caso después del parámetro pondremos una interrogación y en la función se declarará la variable PHP como null.

```
Route::get('/bienvenido/{nombre?}', function ($name = null) {
```

```
    return "Bienvenido " . $name;
});
```

Controladores

Cuando hablamos del modelo vista controlador mencionamos brevemente que el controlador es el mediador entre la fuente de datos (base de datos) y la vista, interfaz gráfica que realmente maneja el usuario.

En Laravel los controladores se encuentran dentro de la carpeta **Controllers**, ubicada dentro de **Http**, que a su vez está dentro de **app**. Podemos ver que en esta carpeta ya hay algunos controladores por defecto, y aquí crearemos los que necesite nuestra aplicación.

Controller.php, es el controlador por defecto del Framework Laravel. Si abrimos el fichero veremos que tiene el siguiente código.

```
<?php

namespace App\Http\Controllers;

use Illuminate\Foundation\Auth\Access\AuthorizesRequests;
use Illuminate\Foundation\Bus\DispatchesJobs;
use Illuminate\Foundation\Validation\ValidatesRequests;
use Illuminate\Routing\Controller as BaseController;

class Controller extends BaseController
{
    use AuthorizesRequests, DispatchesJobs, ValidatesRequests;
}
```

Todos los controladores que creemos tendrán que heredar de esta clase **Controller** para poder adquirir la funcionalidad que ya nos proporciona Laravel en este controlador por defecto.

En el código anterior lo primero que nos encontramos es `namespace App\Http\Controllers;` Namespace como su nombre indica es el espacio de nombres y se utiliza para no crear conflicto con otras clases, métodos o variables que vaya creando el desarrollador. No puede haber dentro de la misma aplicación dos clases o métodos con el mismo nombre salvo que se usen los namespace.

Un namespace es como si fuera una carpeta en el sistema operativo. Ya sabemos que no puede haber dos archivos en la misma carpeta con el mismo nombre, pero sí podemos tener dos archivos con el mismo nombre en carpetas diferentes. Los namespace hacen algo

parecido. Así cuando creamos un archivo PHP y como primera instrucción tenemos un namespace y una ruta, lo que estamos diciendo a nuestra aplicación es que ese espacio de nombre es una especie de carpeta dónde estamos creando una serie de clases, métodos y variables y el ámbito de esos nombres, para que no colisionen con otros, es ese espacio de nombres. Así por ejemplo podríamos crear otra clase Controller siempre que el namespace fuera diferente.

Después tenemos la palabra clave `use`, que sirve para importar diferentes clases que se encuentran en bibliotecas. Son clases que se usarán posteriormente en el código.

¿Cómo podemos crear nuestros propios controladores?

Podemos crearlo de dos formas:

- De forma manual.
- A través de la consola usando los comandos correspondientes.

Controlador de forma manual

Creamos un nuevo archivo PHP dentro de la carpeta Controller. El nombre de este archivo por convención suele ser el nombre que nosotros queramos seguido de la palabra Controller al ser una clase de tipo controlador.

En el archivo creado usaremos el mismo namespace anterior y crearemos una clase con el mismo nombre del fichero. Así si por ejemplo creamos el fichero EjemploController el contenido del fichero sería

```
<?php

namespace App\Http\Controllers;

class EjemploController extends Controller
{

}
```

Como primer ejemplo vamos a crear un controlador que sea capaz de enrutar nuestra aplicación Laravel. Lo mismo que hicimos con las rutas lo vamos a hacer usando los controladores.

Si tenemos pocas páginas en nuestra aplicación podemos usar las rutas como las creamos anteriormente para que nos vayan llevando a esas páginas. Sin embargo si tenemos aplicaciones con muchas páginas en las que tenemos que crear muchas rutas es aconsejable utilizar los controladores para gestionar estas rutas. Podemos usar los

controladores para agrupar estas rutas. ¿Cómo lo hacemos? Para ello creamos un método dentro de nuestra clase

```
<?php

namespace App\Http\Controllers;

class EjemploController extends Controller
{
      public function inicio(){
            return "Estás en el inicio del sitio.";
      }
}
```

Con esto ya tendríamos creado nuestro controlador y ahora vamos a ver como enlazarlo con las rutas.

Abrimos de nuevo el archivo web.php, y en lugar de la función anónima ponemos el nombre del controlador y método al que queramos llamar.

```
Route::get('/inicio', 'EjemploController@inicio');
```

Controlador a través de la consola

Abrimos una consola y nos situamos en la carpeta de laravel y ejecutamos el siguiente código

```
php artisan make:controller Ejemplo2Controller
```

El último parámetro es el nombre del controlador. Si vamos a nuestra carpeta de Controller veremos que se ha creado un nuevo archivo con el siguiente código

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class Ejemplo2Controller extends Controller
{
    //
```

```
}
```

A diferencia del controlador manual importa una clase Request que se suele usar.

Ya sólo quedaría añadir un método como en el caso anterior y enlazarlo con las rutas.

El comando php artisan nos permite usar un montón de opciones. Si en nuestra consola ejecutamos lo siguiente nos permite ver todas las opciones disponibles.

```
php artisan
```

A la instrucción que nos permite crear un controlador podemos pasarle algún argumento. Por ejemplo, podemos crear un controlador para la gestión de un CRUD, que crea un controlador con una serie de métodos ya definidos.

```
php artisan make:controller --resource Ejemplo3Controller
```

Si creamos un controlador con el parámetro resource no es necesario en nuestro fichero de rutas crear una ruta para cada uno de los métodos de este controlador. Para el controlador del ejemplo anterior bastaría tener la siguiente línea.

```
Route::resource('/ejemplo3', 'Ejemplo3Controller');
```

Podemos ver las rutas habilitadas en nuestro proyecto de Laravel a través del siguiente comando.

```
php artisan route:list
```

Creación de controladores y rutas del proyecto de Centros y Departamentos

Para este proyecto crearemos los dos dos controladores a través de la línea de comandos y con el parámetro resources:

Para crear el controlador de Centros

```
php artisan make:controller --resource CentrosController
```

Para crear el controlador de Departamentos

```
php artisan make:controller --resource DepartamentosController
```

Nuestro fichero de ruta **routes.php** tendrá el siguiente aspecto:

```
<?php

/*
|--------------------------------------------------------------------------
| Web Routes
|--------------------------------------------------------------------------
|
| Here is where you can register web routes for your application. These
| routes are loaded by the RouteServiceProvider within a group which
| contains the "web" middleware group. Now create something great!
|
*/

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

Route::resource('/centros', 'CentrosController');
Route::resource('/departamentos', 'DepartamentosController');
```

Enlace a documentación de Laravel para la gestión de rutas. Rutas en Laravel
Enlace a la documentación de Laravel sobre controladores: Controladores en Laravel

Consultas SQL

En Laravel podemos ejecutar consultas SQL directamente en los métodos de nuestro controladores para obtener los datos. Para ello hay que hacer uso de la clase DB (Illuminate\Support\Facades\DB) y en este caso no necesitamos recurrir a los modelos.

Para sacar un listado de todos los centros el código sería

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;

class CentrosController extends Controller
```

```
{
     /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
      //

       $centros = DB::select('select * from centros');
       return view("centros.index",compact("centros"));
    }
```

Si quisiera hacer una consulta con una condición, se podría hacer pasando la consulta junto con la condición en el primer parámetro de la función **select**. De todas formas la forma correcta sería es usando el carácter ? en la consulta y después la función select tener un segundo parámetro que es un array donde se sustituyen los ? por los elementos del array.

Para sacar un centro cuyo id es el 1 tendríamos:

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;

class CentrosController extends Controller
{
     /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
      //
       $centros = DB::select('select * from centros where id=?',
[1]);
       return view("centros.index",compact("centros"));
    }
```

Al igual que podemos hacer consultas para mostrar datos, podemos hacer consultas para insert, actualizar o borrar registros.

Para crear un nuevo centro con nombre “SEDE” y dirección “Av. Hitasa”:

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;

class CentrosController extends Controller
{
     /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
      //
       DB::insert('insert into centros (nombre, direccion) values
(?, ?)', ['SEDE', 'Av. Hitasa'])
    }
```

Para actualizar la dirección del centro 1 a “Av. Andalucía”:

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;

class CentrosController extends Controller
{
     /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
      //
```

```
      DB::update('update set centros direccion=? where id=?',
['Av. Andalucía',1])
    }
```

Para borrar todos los centros:

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;

class CentrosController extends Controller
{
     /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
      //
       DB::delete('delete from centros'])
    }
```

ORM Eloquent

Eloquent es el ORM que incluye Laravel para manejar de una forma fácil y sencilla los procesos correspondientes al manejo de bases de datos. Gracias a las funciones que provee podremos realizar complejas consultas y peticiones de base de datos sin escribir una sola línea de código SQL. Se puede acceder a la documentación de Eloquent en el siguiente enlace: Eloquent

Un ORM (Object Relational Mapping) es un mapeo de objetos relacionales. Básicamente lo que hace es utilizar la programación orientada a objetos para manipular una base de datos. ORM no es una característica de la Laravel, está incluido en muchos otros frameworks.

Vamos a ver esto con las tablas de nuestro proyecto:

- Centros
- Departamentos

Supongamos queremos manipular la tabla “Centros”. Eloquent lo que hace es crear un Modelo basado en la tabla con la que queremos trabajar. Este modelo por tanto representa

a la tabla que queremos manipular. Este modelo no es más que una clase de PHP. A la hora de crear este modelo hay que seguir una nomenclatura muy concreta.

Así para manipular la tabla “Centros” tenemos que crear un modelo que tenga el mismo nombre pero en singular y que empiece por mayúsculas. Este modelo es una clase, un objeto, y por tanto va a tener unas propiedades y métodos que nos va a permitir manipular la tabla sin necesidad de escribir instrucciones SQL.

Para manipular por tanto todas las tablas de nuestro proyecto crearemos un modelo por tabla.

Para crear un modelo de nuestra Centro ejecutaremos

```
php artisan make:model Centro
```

El archivo se crea directamente en la carpeta app.

Si abrimos el fichero Centro.php vemos que es una clase de PHP que hereda de Model. En esta clase Model están todos los métodos y propiedades para poder manipular nuestra base de datos.

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Centro extends Model
{
    //
}
```

Nombre de tabla

Por convención, el nombre de la clase en plural será usado como el nombre de tabla, a menos que otro nombre sea especificado expresamente. Así, aunque no es nuestro caso no es necesario ya que nuestra tabla se llama Centros, se puede especificar el nombre de la tabla con la propiedad protegida **$table** dentro del modelo:

```
<?php

namespace App;
```

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

class Centro extends Model
{
    protected $table = 'nombre_tabla';
}
```

Clave primaria

Eloquent asumirá que cada tabla tiene una columna de clave primaria denominada **id**. Puedes definir una propiedad protegida **$primaryKey** para sobrescribir esta convención en caso de que la clave primaria de la tabla no sea id. En nuestro caso esto tampoco será necesario.

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Centro extends Model
{
    protected $primaryKey = 'clave_primaria';
}
```

Además, Eloquent asume que la clave primaria es un valor entero autoincremental, lo que significa que de forma predeterminada la clave primaria será convertida a un tipo int automáticamente. Si deseas usar una clave primaria que no sea de autoincremento o numérica debes establecer la propiedad pública **$incrementing** de tu modelo a false:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Centro extends Model
{
    public $incrementing = false;
}
```

Si tu clave primaria no es un entero, debes establecer la propiedad protegida **$keyType** de tu modelo a string:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Centro extends Model
{
    protected $keyType = 'string';
}
```

Marcas de tiempo o timestamps

De forma predeterminada, Eloquent espera que las columnas **created_at** y **updated_at** existan en tus tablas. Si no deseas que estas columnas sean manejadas automáticamente por Eloquent, establece la propiedad **$timestamps** de tu modelo a false:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Centro extends Model
{
    public $timestamps = false;
}
```

Si necesitas personalizar el formato de tus marcas de tiempo, establece la propiedad **$dateFormat** de tu modelo. Esta propiedad determina cómo los atributos de fecha son guardados en la base de datos, además de su formato cuando el modelo es serializado a un array o JSON:

```
<?php

namespace App;
```

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

class Centro extends Model
{
    protected $dateFormat = 'U';
}
```

Si necesitas personalizar los nombres de las columnas usadas para guardar las marcas de tiempo, puedes establecer las constantes CREATED_AT y UPDATED_AT en tu modelo:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Centro extends Model
{
    const CREATED_AT = 'creation_date';
    const UPDATED_AT = 'last_update';
```

Definición de las relaciones en los modelos con Eloquent

Ahora vamos a abordar la parte de las relaciones de uno a muchos (1 a N),. Vamos a ver como crear convenientemente los modelos para que informemos al framework de la existencia de esta relación.

Para definir las relaciones tendremos que especificar métodos con una forma concreta, en los que informamos sobre las relaciones. Como la relación afecta a dos tablas, tendremos dos modelos con los que trabajar, en nuestro proyecto las tablas **Centros** y **Departamentos**.

Relación 1 a muchos en el modelo Centro

Comenzamos con la relación que tenemos en la parte del "1", es decir en la tabla que está relacionada con "N" elementos de la otra parte. En nuestro caso es la tabla **Centros** la que tiene el "1" y que está relacionada con "N" departamentos. Decimos que 1 centro tiene muchos departamentos.

Para definir la relación tenemos que crear un método en el modelo, con el nombre que le queramos otorgar a dicha relación, que usualmente será el nombre de la entidad que

queremos relacionar, en este caso en plural, dado que un centro puede relacionarse con muchos departamentos.

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Centro extends Model
{

    public function departamentos()
    {
      return $this->hasMany('App\Departamento');
    }
}
```

El método departamentos() es el que implementa la relación. En él tenemos que devolver el valor de retorno del método hasMany() de los modelos Eloquent. A hasMany le tenemos que informar con el nombre de la clase del modelo con el que estamos relacionando.

Laravel entenderá automáticamente que en la tabla **Departamentos** existirá la clave foránea con el **Centro** (centro_id en la tabla departamento) y que en la tabla local (Centro), la clave primaria se llama "id". Es importante que respetemos las convenciones de nombrado de tablas y de claves foráneas, para que no tengamos que hacer más trabajo al definir las relaciones, pero si no es el caso, hasMany() también puede recibir como parámetros las personalizaciones en las tablas que sean necesarias.

Por ejemplo, si en la tabla departamentos la clave foránea tuviera otro nombre distinto de centro_id, podríamos indicarlo así:

```
return $this->hasMany('Departamento', 'nombre_clave_foranea');
```

Y si fuera el caso que en la tabla de centros la clave primaria no se llamara "id" también podríamos indicarlo con esta llamada a hasMany():

```
return    $this->hasMany('Departamento',    'nombre_clave_foranea',
'nombre_clave_primaria_local');
```

Acceso a los datos de la tabla relacionada

Una vez que tenemos en nuestro modelo definida la relación, podemos acceder a los datos de la tabla relacionada en cualquier modelo de Centro. Para ello usamos el nombre del método que hemos creado como relación, en este caso era "departamentos".

```
$departamentos_de_un_centro = Centro::find(1)->departamentos;
```

Esto será una colección de departamentos que podremos usar como cualquier otra collection de Laravel.

```
foreach($departamentos_de_un_centro as $departamento) {
      // hacer lo que necesites para cada $departamento
}
```

En Laravel el acceso a los datos de las tablas relacionadas se realiza por "lazy load", lo que quiere decir que, hasta que no se acceda a estos campos relacionados, no se hará la correspondiente consulta para la relación. Pero nosotros podemos forzar a Eloquent a que nos traiga de antemano los datos relacionados.

```
$centros = Centro::with('departamentos')->get();
```

La inversa de la relación 1 a N, en el modelo Departamento

Al definir el modelo Departamento también podemos, si se ve necesario, definir la relación. Con ello conseguimos que sea muy sencillo acceder desde el modelo marcado con la "N" a los datos del modelo marcado con el "1". O sea, para nuestro ejemplo, acceder al centro de un determinado departamento.

Esto se consigue con la definición de un método en el modelo, que llevará el nombre que queramos, pero usualmente será el nombre de la entidad que queremos relacionar. En este caso en singular, ya que solo estamos relacionando con un elemento de la otra tabla.

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Departamento extends Model
{
    //
```

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

El método que recibe la relación 1 a N inversa contiene la devolución del método belongsTo(), en el que indicamos el nombre del modelo con el que estamos relacionando.

Del mismo modo que en el caso anterior, es importante que las tablas estén creadas con las convenciones que asume Eloquent. En la tabla departamento se entiende que el nombre del campo de la clave foránea se llama "centro_id" y que el nombre de la clave primaria de la tabla relacionada es "id". Si no fuera el caso tenemos que aleccionar a Eloquent indicando como parámetros los nombres que se han utilizado en la definición de las tablas.

```
return $this->belongsTo('Centro', 'nombre_clave_foranea',
'nombre_clave_otra_tabla');
```

Obteniendo modelos

Una vez creada una tabla y su modelo asociado, estás listo para empezar a obtener datos de tu base de datos. Piensa en cada modelo de Eloquent como un constructor de consultas muy poderoso que te permite consultar la tabla de base de datos asociada con el modelo. Por ejemplo para obtener todos los centros:

Obtener todo los Centros en el método index de mi controlador

```
<?php

namespace App\Http\Controllers;
use Illuminate\Http\Request;
use App\Centro;

class CentrosController extends Controller
{
     /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
```

```
        //
        $centros = Centro::all();
        return view("centros.index",compact("centros"));
    }
```

Añadiendo restricciones adicionales

El método **all** de Eloquent devolverá todos los resultados en la tabla del modelo. Ya que cada modelo de Eloquent sirve como un constructor de consultas, también puedes añadir restricciones a las consultas y entonces usar el método get para obtener los resultados.

Si por ejemplo queremos obtener los 10 primeros departamentos con presupuesto igual a 1000 y ordenados por nombre el código sería:

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
use App\Departamento;

class DepartamentosController extends Controller
{
     /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        //
        $departamentos = Departamento::where("presupuesto",1000)
           ->orderBy('nombre','desc')
           ->take(10)
           ->get();
                                                                return
view("departamentos.index",compact("departamentos"));
    }
```

Si la función **where** sólo lleva dos parámetros se buscarán registros que coincidan con la condición. Si queremos que la condición no sea de igualdad, la función **where** tiene que llevar tres parámetros, siendo el segundo la condición.

Si por ejemplo queremos obtener los 10 primeros departamentos con presupuesto menor o igual a 100 y ordenados por nombre el código sería:

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
use App\Departamento;

class DepartamentosController extends Controller
{
     /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        //
        $departamentos =
Departamento::where("presupuesto","<=",100)
           ->orderBy('nombre','desc')
           ->take(10)
           ->get();
                                                                    return
view("departamentos.index",compact("departamentos"));
    }
```

Si por ejemplo queremos obtener los sólo el primer departamento con presupuesto menor o igual a 100 y ordenado por nombre el código sería:

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
use App\Departamento;

class DepartamentosController extends Controller
{
     /**
     * Display a listing of the resource.
```

```
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        //
        $departamento =
Departamento::where("presupuesto","<=",100)
           ->orderBy('nombre','desc')
           ->first();
                                                                    return
view("departamentos.index",compact("departamento"));
```

En este caso no se devuelve un array de objetos.

Si ni siquiera necesita una fila entera, puede extraer un único valor de un registro utilizando el método del value.

Por ejemplo para extraer sólo la columna nombre del listado de departamentos con presupuesto igual a 1000 el código sería:

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
use App\Departamento;

class DepartamentosController extends Controller
{
     /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        //
        $departamentos = Departamento::where("presupuesto",1000)
           ->value('nombre')
                                                                    return
view("departamentos.index",compact("departamentos"));
```

También se puede hacer uso de funciones SQL.

Por ejemplo para obtener el departamento con mayor presupuesto (función max)

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
use App\Departamento;

class DepartamentosController extends Controller
{
      /**
      * Display a listing of the resource.
      *
      * @return \Illuminate\Http\Response
      */
     public function index()
     {
         //
         $departamento = Departamento::max("presupuesto");
                                                                  return
view("departamentos.index",compact("departamento"));
```

Para obtener el departamento con menor presupuesto (función min)

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
use App\Departamento;

class DepartamentosController extends Controller
{
      /**
      * Display a listing of the resource.
      *
      * @return \Illuminate\Http\Response
      */
     public function index()
     {
```

```
        //
        $departamento = Departamento::min("presupuesto");
        return view("departamentos.index",compact(
```

Ya que los modelos de Eloquent son constructores de consultas, se puede revisar todos los métodos disponibles en el constructor de consultas y usar cualquiera de estos métodos en las consultas de Eloquent (Database Query Builder).

Insertando modelos y actualizando modelos

Para agregar un nuevo registro en la base de datos crea una nueva instancia de modelo, establece los atributos del modelo y después ejecuta el método **save**:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Centro;

class CentrosController extends Controller
{
     public function store(Request $request){

          $centro = new Centro;
          $centro->nombre = 'SEDE ESTE';
          $centro->direccion =  'Paseo de la Reina, Valencia';
          $centro->save();
     }
}
```

En este ejemplo, asignamos cadenas de texto tanto al atributo nombre como al atributo direccion del objeto que acabamos de crear. Cuando veamos las vistas y la solución completa los datos se recibirán de la petición ($request)

Cuando ejecutamos el método save, un nuevo registro será insertado en la base de datos. Las marcas de tiempo created_at y updated_at serán automáticamente establecidas cuando el método save sea ejecutado (si las tenemos establecidas), no hay necesidad de establecerlos manualmente.

Actualizando modelos

El método save también puede ser usado para actualizar modelos que ya existen en la base de datos. Para actualizar un modelo, debes obtenerlo, establecer cualquiera de los atributos que desees actualizar y después ejecutar el método save. Otra vez, la marca de tiempo updated_at será actualizada automáticamente, no hay necesidad de establecer su valor manualmente.

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Centro;

class CentrosController extends Controller
{
      public function update(Request $request, $id){

            $centros = Centro::find(1);
            $centros->direccion =  'Paseo de la Victoria, Valencia';
            $centros->save();
      }
}
```

Asignación masiva

Se puede usar el método **create** para guardar un nuevo modelo en una sola línea. La instancia de modelo insertada será devuelta por el método. Sin embargo, antes de hacer eso, necesitarás especificar o un atributo **fillable** o **guarded** del modelo, de modo que todos los modelos de Eloquent se protejan contra la asignación masiva de forma predeterminada.

Así que, para empezar, debes definir cuáles atributos del modelo quieres que se asignen de forma masiva. Puedes hacerlo usando la propiedad protegida **$fillable** del modelo. Por ejemplo, vamos a hacer que el atributo nombre y dirección de nuestro modelo Centro sea asignado masivamente:

```
<?php

namespace App;
```

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

class Centro extends Model
{
    protected $fillable = ['nombre', 'direccion'];
...
```

Una vez que hemos indicado los atributos asignables en masa, podemos usar el método create para insertar un nuevo registro en la base de datos. El método create devuelve la instancia de modelo guardada:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Centro;

class CentrosController extends Controller
{
     public function store(Request $request){

          $centro = Centro::create(['nombre' => 'SEDE ESTE',
'direccion' => 'Paseo de la Reina']);
          $centro->save();
     }
}
```

Si ya tienes una instancia del modelo, puedes usar el método fill para llenarla con un array de atributos:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Centro;

class CentrosController extends Controller
```

```
{
      public function asignacionMasiva(){

            $centro = new Centro;
            $centro->fill(['nombre' => 'SEDE ESTE', 'direccion' =>
'Paseo de la Reina']);
            $centro->save();
      }
}
```

Actualizaciones masivas

Las actualizaciones también pueden ser ejecutadas contra cualquier número de registros de un modelo que coincidan con un criterio de consulta dada. En este caso al igual que en la asignación masiva tenemos que definir la propiedad protegida **$fillable** en el modelo.

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Departamento extends Model
{
    //
    protected $fillable = ['presupuesto'];
...
```

Una vez que hemos indicado los atributos asignables en masa, podemos usar el método update para actualizar registros en la base de datos

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Departamento;

class DepartamentosController extends Controller
{
```

```
      public function actualizarMasiva(){

$departamentos->where('presupuesto',100)->update(['presupuesto'=>2
00]);
      }
}
```

En este ejemplo, todos los departamentos que tengan un presupuesto de 100 se actualizarán a 200:

El método update espera un array de pares de clave valor, representado por las columnas que deben ser actualizadas.

Eliminando modelos

Para eliminar un modelo, hay que ejecutar el método delete en una instancia del modelo:

```
<?php

...
public function borrar(){
      $centro = new Centro::find(5);
      $centro->delete();
}
...
```

Eliminando un modelo existente por clave

En el ejemplo anterior, estamos obteniendo el modelo de la base de datos antes de ejecutar el método delete. Sin embargo, si conoces la clave primaria del modelo, puedes eliminar el modelo sin obtenerlo primero. Para hacer eso, ejecuta el método **destroy**. Además de recibir una sola clave primaria como argumento, el método destroy aceptará múltiples claves primarias, un array de claves primarias, o una colección de claves primarias:

```
<?php

...
public function borrar(){
    Centro::destroy(5);
}
```

```
...
```

Eliminando modelos por consultas

También puedes ejecutar una instrucción de eliminar en un conjunto de modelos. En este ejemplo, eliminaremos todos los departamentos que tengan presupuesto de 200.

```
<?php

...
public function borrar(){
  $deletedRows = Departamentos::where('presupuesto',
200)->delete();
);

}
```

Eliminación lógica (Soft Deleting)

Además de eliminar realmente los registros de tu base de datos, Eloquent también puede eliminar lógicamente los modelos. Cuando los modelos son borrados lógicamente, no son eliminados realmente de la base de datos. En lugar de eso, un atributo deleted_at es establecido en el modelo e insertado en la base de datos. Si un modelo tiene un valor deleted_at no nulo, el modelo ha sido eliminado lógicamente.

Para habilitar eliminaciones lógicas en un modelo, hay que usar la clase usa SoftDeletes (Illuminate\Database\Eloquent\SoftDeletes) en el modelo:

```
<?php

namespace App;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;

class Centro extends Model
{
    //
     use SoftDeletes;
     protected $fillable = ['nombre', 'direccion'];
     public $timestamps = false;
```

```
}
```

También debe estar añadida la columna **deleted_at** a tu tabla de base de datos. Para añadir esa columna a la tabla bastaría con haber puesto en el fichero de migración la siguiente línea:

```
$table->softDeletes();
```

Ahora, cuando se ejecuta el método delete en el modelo, la columna deleted_at es establecida con la fecha y hora actual. Y, al momento de consultar un modelo que use eliminaciones lógicas, los modelos eliminados lógicamente serán excluidos automáticamente de todos los resultados de consultas.

Para determinar si una instancia de modelo ha sido eliminada lógicamente, usa el método trashed:

```
if ($centro->trashed()) {
    //
}
```

Consultando modelos eliminados lógicamente

Incluyendo modelos eliminados lógicamente

Como se apreció anteriormente, los modelos eliminados lógicamente serán excluidos automáticamente de los resultados de las consultas. Sin embargo, puedes forzar que los modelos eliminados lógicamente aparezcan en un conjunto resultante usando el método withTrashed en la consulta:

```
$centros = Centro::withTrashed()->get();
```

Obteniendo modelos individuales eliminados lógicamente

El método onlyTrashed obtendrá solamente modelos eliminados lógicamente:

```
$centros = Centro::onlyTrashed()->get();
```

Restaurando modelos eliminados lógicamente

Algunas veces se quiere deshacer la eliminación de un modelo eliminado lógicamente. Para restaurar un modelo eliminado lógicamente a un estado activo, hay que usar el método restore en una instancia de modelo:

```
$centro->restore
```

También se puede usar el método restore en una consulta para restaurar rápidamente varios modelos.

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Departamento;

class DepartamentosController extends Controller
{
      public function restaurarMasiva(){

$departamentos::withTrashed()->where('presupuesto',100)->restore()
;
      }
}
```

Vistas

Las vistas en Laravel son la parte pública que el usuario de nuestro sistema va a poder ver, se escriben en HTML junto con un motor de plantillas llamado Blade. Las vistas se encuentran ubicadas en la carpeta **resources/views/** y Laravel por defecto trabaja con la idea de que tenemos que escribir la menor cantidad de código repetido, modularizar nuestro código en donde más se pueda. El motor de plantillas de Blade utilizan una nomenclatura que empieza por @. Blade dispone de muchas etiquetas o directivas y toda la documentación la podemos encontrar en el siguiente enlace: Motor de plantillas de Laravel

Una vista puede ser devuelta tanto desde el fichero de rutas, **web.php** (return view('welcome');) como desde un método de un controlador. A las vistas se le pueden pasar datos y a continuación vamos a ver cómo..

Podemos pasar datos a la vista mediante un array asociativo, donde las claves son el nombre de las variables que queremos pasar a la vista y el valor son los datos que queremos asociar:

```
public function quienesSomos (){
```

```
      $nombre = 'Antonio';
      return view('quienesSomos', ['name' => $nombre]);
}
```

En la vista para poder recibir el parámetro tenemos que poner el parámetro entre llaves

```
{{ $name }}
```

También podemos usar el método **with** encadenando en la llamada a la función view.

```
public function quienesSomos (){
      $nombre = 'Antonio';
      return view('quienesSomos')->with(['name', $nombre]);
}
```

Con **with** también podemos pasar las variables de forma individual:

```
public function quienesSomos (){
      return view('quienesSomos')->with('name' => 'Antonio');
}
```

Con **with** también podemos pasar más de una variable de forma individual:

```
public function quienesSomos (){
      return view('quienesSomos')->with('name' =>
'Antonio')->with('title' => 'Título');
}
```

Con **compact** también podemos pasar variables a nuestra vista:

```
public function quienesSomos (){
      $name = 'Antonio';
      $title = 'Título';
      return view('welcome', compact('name', 'title'));
}
```

Imprimir variables

Si queremos imprimir una variable que le llega a nuestra vista, podemos hacerlo utilizando la sintaxis de dobles llaves {{ }}

```
{{ $usuario }}
```

Ciclos y estructuras

Si queremos utilizar bucles y estructuras condicionales, podemos utilizar directivas. Las directivas de Blade van precedidas por un arroba (@) y luego el nombre de la directiva:

```
@foreach ($usuarios as $usuario)
    <li>{{ $usuario }}</li>
@endforeach
```

También podemos utilizar la directiva @if:

```
@if (! empty($usuarios))
    ...
@endif
```

La directiva @if puede ser utilizada junto con un bloque else (utilizando @else):

```
@if (! empty($usuarios))
    ...
@else
    <p>No hay usuarios registrados.</p>
@endif
```

Podemos utilizar la directiva @elseif, que como su nombre sugiere nos permite utilizar un bloque else if:

```
@if (! empty($usuarios))
    ...
@elseif ($users < 3)
    <p>Hay menos de 3 usuarios registrados.</p>
@else
    <p>No hay usuarios registrados.</p>
```

```
@endif
```

Blade también tiene la directiva @unless, que funciona como un condicional inverso::

```
@unless (empty($usuarios))
    <ul>
        @foreach ($usuarios as $usuario)
            <li>{{ $usuario }}</li>
        @endforeach
    </ul>
@else
    <p>No hay usuarios registrados.</p>
@endunless
```

En el ejemplo anterior queremos mostrar el listado de usuarios a no ser que la lista esté vacía. De lo contrario queremos mostrar el mensaje del bloque else.

También podemos utilizar la directiva @empty que es una forma más corta de escribir @if (empty (...))

```
@empty($usuarios)
    <p>No hay usuarios registrados.</p>
@else
    <ul>
        @foreach ($usuarios as $usuario)
            <li>{{ $usuario }}</li>
        @endforeach
    </ul>
@endempty
```

Además de @foreach, también podemos utilizar @for:

```
@for ($i = 0; $i < 10; $i++)
    El valor actual es {{ $i }}
@endfor
```

Con la directiva @forelse podemos asignar una opción por defecto a un ciclo foreach sin utilizar bloques anidados:

```
@forelse ($usuarios as $usuario)
    <li>{{ $usuario }}</li>
@empty
    <li>No hay usuarios registrados.</li>
@endforelse
```

Con la directiva @switch podemos crear una estructura de control tipo switch con sus directivas @case, @break, @default

```
@switch($i)
    @case(1)
        Primer caso...
        @break
    @case(2)
        Segundo caso...
        @break
    @default
        Caso por defecto...
@endswitch
```

Al hacer un bucle, una variable **$loop** estará disponible dentro de tu bucle. Esta variable proporciona acceso a información útil del bucle, como el índice del bucle actual y si esta es la primera o la última iteración del bucle:

```
@foreach ($usuarios as $usuario)
    @if ($loop->first)
        Es la primera iteración.
    @endif

    @if ($loop->last)
        Es la última iteración.
    @endif
@endforeach
```

Si estás en un bucle anidado, puedes acceder a la variable $loop del bucle padre a través de la propiedad parent:

```
@foreach ($usuarios as $usuario)
    @foreach ($usuario->posts as $post)
```

```
        @if ($loop->parent->first)
            Esta es la primera  iteración del primer bucle.
        @endif
    @endforeach
@endforeach
```

La variable $loop también contiene otras propiedades útiles:

Propiedad	Descripción
$loop->index	El índice de la iteración actual del bucle (comienza en 0)
$loop->iteration	La iteración del bucle actual (comienza en 1).
$loop->remaining	Las iteraciones que quedan en el bucle.
$loop->count	El número total de elementos en la matriz que se itera.
$loop->first	Si esta es la primera iteración a través del bucle.
$loop->last	Si esta es la última iteración a través del bucle.
$loop->even	Si esta es una iteración par del bucle.
$loop->odd	Si esta es una iteración impar del bucle.
$loop->depth	El nivel de anidación del bucle actual.
$loop->parent	En un bucle anidado, la variable del bucle padre.

CSRF

Laravel hace que sea fácil proteger tu aplicación de ataques de tipo cross-site request forgery (CSRF). Los ataques de tipo CSRF son un tipo de explotación de vulnerabilidad malicioso por el cual comandos no autorizados son ejecutados en nombre de un usuario autenticado.

Laravel genera automáticamente un «token» CSRF para cada sesión de usuario activa manejada por la aplicación. Este token es usado para verificar que el usuario autenticado es quien en realidad está haciendo la petición a la aplicación.

Cada vez que se defina un formulario HTML en una aplicación, se debe incluir un campo de token CSRF oculto en el formulario para que Laravel pueda validar la solicitud. Este token se crea con la directiva @csrf.

```
<form method="POST" action="/centros">
    @csrf

    ...
</form>
```

Campo método

Dado que los formularios HTML no pueden hacer peticiones PUT, PATCH o DELETE, se tiene que añadir un campo oculto con nombre **_method** oculto para falsificar estos verbos HTTP. La directiva @method Blade permite crear este campo de manera más fácil

```
<input type="hidden" name="_method" value="PUT">
```

Con la directiva @method sería

```
@method("PUT")
```

Validando errores

La directiva @error puede utilizarse para comprobar rápidamente si existen mensajes de error de validación para un atributo determinado. Dentro de una directiva @error, puede hacerse eco de la variable $message para mostrar el mensaje de error:

```
@error('nombre')
    <div>{{ $message }}</div>
@enderror
```

Comentarios

Blade también permite definir comentarios en las vistas. Sin embargo, a diferencia de los comentarios HTML, los comentarios de Blade no se incluyen en el HTML devuelto por la aplicación:

```
{{-- Este comentario no se mostrará en el HTML renderizado --}}
```

Plantillas

A la hora de crear las páginas de un sitio web lo más habitual es que la mayoría de las páginas tengan una estructura similar. Es decir que todas estas páginas comporten una misma cabecera y pie. Si no tuviéramos el motor de plantillas, para no tener que copiar y pegar esas cabeceras y pies en cada una de las páginas, podríamos usar los include/require vistos de PHP.

Con el motor de plantillas de Blade podemos crear una plantilla, también con extensión blade, que integrará los elementos comunes y que se encargará de compartirlos con todos aquellos archivos que hereden de esta plantilla. En las plantillas es muy común hacer uso de la etiqueta @section, que sirve para delimitar una sección.

Vamos a crear una plantilla (layout) de Blade (layout.blade.php)

```
<!DOCTYPE html>
<html lang="{{ str_replace('_', '-', app()->getLocale()) }}">
    <head>
        <meta charset="utf-8">
               <meta  name="viewport"  content="width=device-width,
initial-scale=1">

        <title>Laravel</title>

        <!-- Fonts -->
        <link
href="https://fonts.googleapis.com/css2?family=Nunito:wght@200;600
&display=swap" rel="stylesheet">
        <!-- Styles -->
        <style>
           .cabecera {
                background-color: red;
                text-align: center;
           }
           .cuerpo {
                background-color: blue;
                margin: 100px 0;
                color: white;
           }
           .pie {
                background-color: yellow;
           }
        </style>
```

```
    </head>
    <body>

      <div class="cabecera">

            @yield("cabecera")

      </div>

      <div class="cuerpo">

            @yield("cuerpo")

      </div>

      <div class="pie">

            @yield("pie")

      </div>

      </body>
</html>
```

Para usar esta plantilla en otra vista tendremos que usar la directiva extends, que recibe como parámetro la ruta de la plantilla. En la vista tendremos que declarar una sección (@section) que tendrá como parámetro uno de los nombres de las áreas definidas en nuestra plantilla (@yield).

Si queremos generar una vista que herede de la plantilla y que tenga las áreas de cabecera y cuerpo personalizadas tendrá un aspecto como:

```
@extends("layout")

@section("cabecera")
      <h1>Cabecera de la vista</h1>
@endsection

@section("cuerpo")
      <p>Cuerpo de la vista</p>
@endsection
```

Generar URLs

Laravel provee varios métodos que nos ayudan en la generación de URLs de una aplicación. Estos son principalmente útiles cuando se construyen enlaces en las plantillas que redireccionan a otra parte de su aplicación.

Generar URLs con el helper url

Con este helper el parámetro de "url" es la ruta a la que queremos enlazar.

```
<a href="{{ url('centros') }}">Enlace a Centros</a>
```

Todo lo que está entre las llaves de Blade es una expresión de PHP, por lo tanto podemos utilizar variables para generar URLs un poco más dinámicas. Por ejemplo si estamos dentro de un bucle podemos usar la variable loop para generar distintos enlaces.

```
<a href="{{ url("centros/{$loop->iteration}/editar") }}">
```

Algunos programadores cometen el error de escribir:

```
{{ url("centros/{{ $loop->iteration }}/editar") }}
```

Esto no es válido, lo que está dentro de las llaves es una expresión de PHP y no tiene la sintaxis de Blade.

Generar URLs con el helper route

Otra forma de generar URLs es con el helper route, que acepta como primer argumento el nombre de la ruta y como segundo argumento los parámetros dinámicos de la ruta, en caso de que los tenga.

```
<a href="{{ route('donde.estamos') }}">Donde Estamos</a>
```

Para poder utilizar el helper route necesitamos agregarle un nombre a nuestras rutas. Así el fichero web.php tendremos que tener

```
Route::get('/dondeEstamos',
'PaginasController@dondeEstamos')->name("donde.estamos");
```

Si por ejemplo dentro de nuestro fichero de rutas tenemos una ruta usuario que admite dos parámetros: **nombre** e **id** como indicamos a continuación:

```
Route::get('/usuario/{id}/{nombre}', function ($id, $name) {
    return "Bienvenido " . $name . " :" .$id;
```

```
})->name("usu");
```

En nuestra plantilla blade para construir un enlace a la ruta anterior tendremos que tener:

```
<a href="{{ route('usu',['id'=>1, 'nombre'=>'juan']) }}">Usuario</a>
```

Recursos estáticos

En Laravel las imágenes, hojas de estilos y librerías javascript se incluyen en la carpeta **public** que está al mismo nivel que app.

Lo suyo es organizar las hojas de estilos en una carpeta css, los ficheros javascript en una carpeta js y las imágenes en una carpeta images, pero cada desarrollador tiene libertad para organizarlo a su gusto.

Para incluir estos archivos en las vistas se puede hacer uso de la función **asset** que proporciona Laravel. Así si por ejemplo en nuestra carpeta public creamos una carpeta images y dentro de ella tenemos una imagen con nombre ejemplo.png que queremos mostrar en uno de nuestras vistas tendremos que hacerlo de la siguiente manera.

```
<img src="{{ asset('images/ejemplo.png') }}">
```

Creación de un CRUD en Laravel paso a paso

Creación del proyecto

Desde la consola de MSDOS ejecutaremos el siguiente comando, donde dwes_crud_laravel es el nombre del proyecto que vamos a crear.

```
composer create-project --prefer-dist laravel/laravel dwes_crud_laravel "6.*"
```

Crear la Base de Datos

Antes de empezar a trabajar en un proyecto tendremos que crear la base de datos. En nuestro caso el nombre que vamos a asignar a nuestra base de datos es **dwes_laravel**.

Modificar el archivo .env

Una vez creada la base de datos abriremos el archivo **.env** que se encuentra en la raíz del proyecto. En este archivo tendremos que poner los datos de conexión de nuestra base de datos.

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=dwes_laravel
DB_USERNAME=root
DB_PASSWORD=
```

Creación de tablas con el sistema de migraciones

Crear el fichero de migración de la tabla Centros.

```
php artisan make:migration create_centros_table --create="centros"
```

Abrir el fichero de fichero creado y añadir la definición de los campos

```
<?php

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

class CreateCentrosTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('centros', function (Blueprint $table) {
            $table->bigIncrements('id');
            $table->string('nombre',100);
```

```
            $table->string('direccion');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('centros');
    }
}
```

Crear el fichero de migración de la tabla Departamentos.

```
php artisan make:migration create_departamentos_table --create="departamentos"
```

Abrir el fichero de fichero creado y añadir la definición de los campos y la relación entre las tablas Centros y Departamentos

```
<?php

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

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

```
            Schema::create('departamentos', function (Blueprint
$table) {
            $table->bigIncrements('id');
            $table->string('nombre',100)->unique();
            $table->unsignedInteger('presupuesto');
            $table->unsignedBigInteger('centro_id');
            $table->timestamps();

$table->foreign('centro_id')->references('id')->on('centros');
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('departamentos');
    }
}
```

Tras crear los ficheros de migraciones con la estructura de las tablas hay que trasladarlas a la base de datos. Para ello sólo tendremos que ejecutar el comando:

```
php artisan migrate
```

Creación de los controladores

Crear los controladores de para los CRUD de Centros y Departamentos

Controlador para los Centros

```
php artisan make:controller --resource CentrosController
```

Controlador para los Departamentos

```
php artisan make:controller --resource DepartamentosController
```

Rutas

Abrir el fichero de rutas (webs.php) e incluir las rutas los controladores que acabamos de crear

```
<?php

/*
|--------------------------------------------------------------------------
| Web Routes
|--------------------------------------------------------------------------
|
| Here is where you can register web routes for your application. These
| routes are loaded by the RouteServiceProvider within a group which
| contains the "web" middleware group. Now create something great!
|
*/

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

Route::resource('/centros', 'CentrosController');
Route::resource('/departamentos', 'DepartamentosController');
```

Crear Modelos

Crear los modelos para de las tablas Centros y Departamentos que nos van a permitir interactuar con la base de datos.

Modelo para los Centros

```
php artisan make:model Centro
```

Modelo para los Departamentos

```
php artisan make:model Departamento
```

A continuación modificamos los ficheros Centro.php y Departamento.php para definir la relación entre los modelos

Fichero Centro.php

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
use App\Departamento;

class Centro extends Model
{
    //
    public function departamentos()
    {
      return $this->hasMany('Departamento');
    }
}
```

Fichero Departamento.php

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
use App\Centro;

class Departamento extends Model
{
    //
    public function centro()
    {
        return $this->belongsTo('Centro');
    }
}
```

Creación de la plantilla

A continuación dentro de nuestra carpeta de resources/view crear una carpeta layouts. Dentro de la carpeta layout, crear un fichero plantilla.blade.php con el siguiente contenido.

Fichero plantilla.blade.php

```
<!DOCTYPE html>
<html lang="{{ str_replace('_', '-', app()->getLocale()) }}">
    <head>
        <meta charset="utf-8">
               <meta  name="viewport"  content="width=device-width,
initial-scale=1">

        <title>Laravel CRUD</title>

        <!-- Fonts -->
                                                                <link
href="https://fonts.googleapis.com/css2?family=Nunito:wght@200;600
&display=swap" rel="stylesheet">

        <style>

            .cabecera{
                text-align: center;
                font-size: x-large;
                margin-bottom: 100px;
                color: blue;
            }

            .cuerpo form, .cuerpo table{
                width: 400px;
                margin: 0 auto;
            }

            .pie{
                position: fixed;
                bottom: 0px;
                width: 100%;
                font-size: 0.7em;
```

```
                margin-bottom: 15px;
            }

            .centros, .departamentos {
                width: 200px;
                text-align: center;
                font-size: 2em;

            }
        </style>
    </head>
    <body>
        <a href="{{ url('') }}">Inicio</a>
     <div class="cabecera">
          @yield("cabecera")

     </div>

     <div class="cuerpo">

          @yield("cuerpo")

     </div>

     <div class="pie">
           IES Velázquez
          @yield("pie")
     </div>

     </body>
</html>
```

Después de crear la plantilla crearemos la página de inicio que simplemente tendrá dos enlaces a los centros y a los departamentos

```
@extends("layouts.plantilla")
```

```
@section("cuerpo")
    <div style="display: flex">
        <div class="centros">
            <a href="{{ route('centros.index')}}">Centros</a>
        </div>
        <div class="departamentos">
            <a href="{{
route('departamentos.index')}}">Departamentos</a>
        </div>
    </div>
@endsection

@section("pie")
@endsection
```

Tras crear la plantilla y la página de inicio crearemos dos nuevas carpetas de resources/view, una con el nombre “centros” donde pondremos las vistas de los centros y otra carpeta “departamentos” donde pondremos las vistas de los departamentos.

Alta de un Centro

Para gestionar el alta de un centro lo primero que tendremos que hacer es crear una vista create.blade.php dentro de la carpeta centros con el siguiente contenido

```
@extends("../layouts.plantilla")

@section("cabecera")
     Insertar un nuevo Centro
@endsection

@section("cuerpo")
     <form method="post" action="/centros">
          @csrf
          <table>
               <tr>
                    <td>Nombre</td>
                    <td><input type="text" name="nombre"></td>
               </tr>
               <tr>
                    <td>Dirección</td>
```

```
                    <td><input type="text" name="direccion"></td>
                </tr>
                <tr>
                    <td     colspan="2"     align="center"><input
type="submit" name="enviar" value="Enviar"></td>
                </tr>
            </table>
     </form>
     @if ($errors->any())
    <div>
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
     @endif
@endsection

@section("pie")
@endsection
```

A continuación tenemos que modificar el método **create** del controlador CentrosController para que devuelva la vista que acabamos de crear

Método create del controlador CentrosController

```
...
     * Show the form for creating a new resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function create()
    {
        //
        return view("centros.create");
    }
...
```

También tenemos que modificar el método **store** para que guarde el centro rellenado en el formulario.
Importante: Recordad que el modelo debe estar incluido en nuestro controlador para que no de error

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Centro;
...
```

Método store del controlador CentrosController

```
...
     /**
     * Store a newly created resource in storage.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
       $validaciones = ['nombre' => 'required', 'direccion' =>
'required'];
       $mensajes = ['nombre.required' => 'El campo :attribute no
puede estar vacío.', 'direccion.required' => 'El campo :attribute
no puede estar vacío.'];

        $this->validate($request, $validaciones, $mensajes);

        $centro = new Centro;
        $centro->nombre = $request->nombre;
        $centro->direccion = $request->direccion;
        $centro->save();

        return redirect('/centros');
```

```
    }
...
```

Lo primero que encontramos en el método store son dos array. Un primer array **$validaciones** donde indicamos que tanto el campo nombre como el campo dirección son obligatorios y un array **$mensajes** donde personalizamos los mensajes de error.
Tras estos dos arrays tenemos una función validate que recibe tres parámetros, el primero es el request, el segundo el array de validaciones y el tercero es un array con los mensajes de validación personalizados. Si no pasamos este tercer array, se mostrarán los mensajes de validación por defecto de Laravel, que aparecen en inglés.

Si no se cumplen las validaciones, en la directiva @errors tenemos los errores y pueden ser mostrados en nuestra vista.
El código en nuestra vista de create.blade.php que muestra los errores es

```
...
     @if ($errors->any())
    <div>
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
     @endif
...
```

Si no hay errores en nuestro controlador, se creará el centro y se redireccionará al método index del controlador.

Listado de los Centros

En el método index de nuestro controlador lo que vamos a hacer es mostrar el listado de todos los centros.

Método index del controlador CentrosController

```
...
     /**
     * Display a listing of the resource.
     *
```

```
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        //
        $centros = Centro::all();

        return view("centros.index",compact("centros"));
    }
...
```

Tendremos también que crear una vista, para mostrar todos los centros.

Vista index.blade para mostrar todos los centros

```
@extends("../layouts.plantilla")

@section("cabecera")
     Listado de Centros
@endsection

@section("cuerpo")
<table border ="1">
     <tr>
          <th>Nombre</th>
          <th>Acciones</th>
     </tr>


     @foreach ($centros as $centro)
     <tr>
          <td>{{ $centro->nombre }}</td>
          <td align="center"><a href="{{ route('centros.edit',
$centro->id)}}">editar</a> - <a href="{{ route('centros.show',
$centro->id)}}">mostrar</a></td>
     </tr>
     @endforeach
     <tr>
```

```
            <td colspan="2" align="center"><a href="{{
route('centros.create')}}">Nuevo centro</a></td>
      </tr>
</table>

@endsection

@section("pie")
@endsection
```

En esta vista tenemos un enlace para crear un nuevo centro (método create del controlador CentrosController) y otros dos enlaces:

- Para editar un centro que llamará al método edit de CentrosController.
- Para mostrar un centro que llamará al método show de CentrosController.

Estos dos métodos reciben el id del centro que bien queremos editar o bien del que queremos mostrar.

Mostrar información completa de un Centro

Método show del controlador CentrosController

```
...
     /**
     * Display the specified resource.
     *
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function show($id)
    {
        //
        $centros =
Centro::with('departamentos')->where('id',$id)->get();
        return view("centros.show", compact('centros'));
    }
...
```

En el método se usa el método findOrFail para que muestre una página de error 404 en caso de error a la hora de recuperar los datos.

Tendremos también que crear una vista, para mostrar los datos del centro, en este caso nombre, dirección y departamentos del centro.

Vista show.blade para mostrar información del centro seleccionado

```
@extends("../layouts.plantilla")

@section("cabecera")
     Información del Centro
@endsection

@section("cuerpo")
<table border ="1">
     <tr>
          <th>Nombre</th>
          <th>Dirección</th>
          <th>Departamentos</th>
     </tr>
     @foreach ($centros as $centro)
     <tr>
          <td>{{ $centro->nombre }}</td>
          <td>{{ $centro->direccion }}</td>
          <td>
               @if(count($centro->departamentos) == 0)
                    <p>No tiene departamentos asociados.</p>
               @else
                    <ul>
                         @foreach ($centro->departamentos as
$departamento)
                              <li>{{ $departamento->nombre
}}</li>
                         @endforeach
                    </ul>
               @endif
          </td>
     </tr>
     @endforeach
</table>
```

```
@endsection

@section("pie")
@endsection
```

Editar de un Centro

Método edit del controlador CentrosController

```
...
    /**
     * Show the form for editing the specified resource.
     *
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function edit($id)
    {
        //
        $centro = Centro::findOrFail($id);
        return view("centros.edit", compact('centro'));
    }

...
```

Vista edit.blade para mostrar información del centro seleccionado que se quiere editar. En esta vista se ha tenido en cuenta que si el centro tiene asociado departamentos no se puede borrar.

```
@extends("../layouts.plantilla")

@section("cabecera")
     Actualizar un Centro
@endsection

@section("cuerpo")
     <form method="post" action="/centros/{{ $centro->id }}">
          @method("PUT")
          @csrf
```

```
          <table>
               <tr>
                    <td>Nombre</td>
                    <td><input      type="text"      name="nombre"
value="{{ $centro->nombre }}"></td>
               </tr>
               <tr>
                    <td>Dirección</td>
                    <td><input    type="text"    name="direccion"
value="{{ $centro->direccion }}"></td>
               </tr>
               <tr>
                    <td colspan="2" align="center"><input
type="submit" name="enviar" value="Actualizar"></td>
               </tr>
          </table>
     </form>
     <form method="post" action="/centros/{{ $centro->id }}">
          @method("DELETE")
          @csrf
          <input type="submit" name="borrar" value="Eliminar"
          @if (count($centro->departamentos) >0 ) disabled @endif>
     </form>
     @if ($errors->any())
    <div>
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
     @endif

@endsection

@section("pie")
@endsection
```

En la vista de editar en el primer formulario tenemos que añadir un campo, lo hemos hecho a través de la directiva @method, para indicar que la petición es de tipo PUT y así poder ir al método update de nuestro controlador al pulsar el botón actualizar.

Método update del controlador CentrosController

```
...
    /**
     * Update the specified resource in storage.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function update(Request $request, $id)
    {
        //
          $validaciones = ['nombre' => 'required', 'direccion' =>
'required'];
         $mensajes = ['nombre.required' => 'El campo :attribute no
puede estar vacío.', 'direccion.required' => 'El campo :attribute
no puede estar vacío.'];

        $this->validate($request, $validaciones, $mensajes);

        $centro = Centro::findOrFail($id);
        $centro->nombre = $request->nombre;
        $centro->direccion = $request->direccion;
        $centro->save();

        return redirect('/centros');
    }
...
```

En este caso, al igual que en el de creación hemos añadido la validación de que los campos dirección y centros no estuvieran vacíos. Al actualizar el centro se redirecciona de nuevo al listado de centros.

Borrar un Centro

Cuando pulsemos el botón eliminar de nuestra página de edición de un centro vamos al método destroy del controlador CentrosController

Método destroy del controlador CentrosController

```
...
    /**
     * Remove the specified resource from storage.
     *
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function destroy($id)
    {
        //
        $centro = Centro::findOrFail($id);
        $centro->delete();

        return redirect('/centros');
    }
...
```

Controlador y Vistas para los departamentos

Para los departamentos todas las vistas serán creadas dentro de una carpeta como hicimos para los centros. En este caso el nombre de la carpeta será departamentos.

Controlador de departamento (DepartamentosController)

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Validation\Rule;
use App\Centro;
use App\Departamento;

class DepartamentosController extends Controller
{
```

```
    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        //
        $departamentos = Departamento::all();

                                                                    return
view("departamentos.index",compact("departamentos"));
    }

    /**
     * Show the form for creating a new resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function create()
    {
        //
        $centros = Centro::all();
        return view("departamentos.create", compact("centros"));
    }

    /**
     * Store a newly created resource in storage.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        //
                                $validaciones    =    ['nombre'    =>
'required|unique:departamentos|max:100',      'presupuesto'      =>
'required', 'centro_id' => 'required'];
```

```
        $mensajes = ['nombre.required' => 'El campo :attribute no puede estar vacío.', 'nombre.unique' => 'Ese :attribute ya está dado de alta.', 'nombre.max' => 'El campo :attribute no puede tener más de :max caracteres.', 'presupuesto.required' => 'El campo :attribute no puede estar vacío.', 'centro_id.required' => 'Debe seleccionar un centro.'];

        $this->validate($request, $validaciones, $mensajes);

        $departamento = new Departamento;
        $departamento->nombre = $request->nombre;
        $departamento->presupuesto = $request->presupuesto;
        $departamento->centro_id = $request->centro_id;
        $departamento->save();

        return redirect('/departamentos');
    }

    /**
     * Display the specified resource.
     *
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function show($id)
    {
        //
        $departamento = Departamento::findOrFail($id);
                            return    view("departamentos.show", compact('departamento'));
    }

    /**
     * Show the form for editing the specified resource.
     *
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function edit($id)
```

```
    {
        //
        $departamento = Departamento::findOrFail($id);
        $centros = Centro::all();
          return view("departamentos.edit", compact('departamento',
'centros'));
    }

    /**
     * Update the specified resource in storage.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function update(Request $request, $id)
    {
        //
                              //$validaciones    =    ['nombre'    =>
'required|unique:departamentos|max:100',      'presupuesto'      =>
'required', 'centro_id' => 'required'];
                                    $validaciones    =['nombre'    =>
['required','max:100',Rule::unique('departamentos')->ignore($id)],
'presupuesto' => 'required', 'centro_id' => 'required'];
         $mensajes = ['nombre.required' => 'El campo :attribute no
puede estar vacío.', 'nombre.unique' => 'Ese :attribute ya está
dado de alta.', 'nombre.max' => 'El campo :attribute no puede
tener más de :max caracteres.', 'presupuesto.required' => 'El
campo :attribute no puede estar vacío.', 'centro_id.required' =>
'Debe seleccionar un centro.'];

        $this->validate($request, $validaciones, $mensajes);

        $departamento = Departamento::findOrFail($id);;
        $departamento->nombre = $request->nombre;
        $departamento->presupuesto = $request->presupuesto;
        $departamento->centro_id = $request->centro_id;
        $departamento->save();

        return redirect('/departamentos');
```

```
    }

    /**
     * Remove the specified resource from storage.
     *
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function destroy($id)
    {
        //
        $departamento = Departamento::findOrFail($id);
        $departamento->delete();

        return redirect('/departamentos');
    }

}
```

Vista create.blade para el alta de un departamento

```
@extends("../layouts.plantilla")

@section("cabecera")
     Insertar un nuevo Departamento
@endsection

@section("cuerpo")
     <form method="post" action="/departamentos">
          @csrf
          <table>
               <tr>
                    <td>Nombre</td>
                    <td><input type="text" name="nombre"></td>
               </tr>
               <tr>
                    <td>Presupuesto</td>
                    <td><input                          type="text"
name="presupuesto"></td>
```

```
                </tr>
                <tr>
                     <td>Centro</td>
                     <td><select name="centro_id">
                               <option value="">Seleccione
Centro</option>
                               @foreach ($centros as $centro)
                               <option value="{{ $centro->id
}}">{{ $centro->nombre }}</option>
                               @endforeach
                          </select>
                     </td>
                </tr>
                <tr>
                     <td colspan="2" align="center"><input
type="submit" name="enviar" value="Enviar"></td>
                </tr>
           </table>
      </form>
      @if ($errors->any())
    <div>
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
      @endif
@endsection

@section("pie")
@endsection
```

Vista index.blade para mostrar los departamentos

```
@extends("../layouts.plantilla")

@section("cabecera")
     Listado de Departamentos
```

```
@endsection

@section("cuerpo")
<table border ="1">
     <tr>
          <th>Departamento</th>
          <th>Acciones</th>
     </tr>

     @foreach ($departamentos as $departamento)
     <tr>
          <td>{{ $departamento->nombre }}</td>
          <td align="center"><a href="{{
route('departamentos.edit', $departamento->id)}}">editar</a> - <a
href="{{ route('departamentos.show',
$departamento->id)}}">mostrar</a></td>
     </tr>
     @endforeach
     <tr>
          <td      colspan="2"      align="center"><a      href="{{
route('departamentos.create')}}">Nuevo departamento</a></td>
     </tr>
</table>

@endsection

@section("pie")
@endsection
```

Vista show.blade para mostrar información de un departamento

```
@extends("../layouts.plantilla")

@section("cabecera")
     Información del Departamento
@endsection

@section("cuerpo")
<table border ="1">
```

```
     <tr>
          <th>Nombre</th>
          <th>Presupuesto</th>
          <th>Centro</th>
     </tr>
     <tr>
          <td>{{ $departamento->nombre }}</td>
          <td>{{ $departamento->presupuesto }}</td>
          <td>{{ $departamento->centro->nombre }}</td>
     </tr>
</table>

@endsection

@section("pie")
@endsection
```

Vista edit.blade para editar un departamento

```
...
    /**
     * Remove the specified resource from storage.
     *
     * @param  int  $id
     * @return@extends("../layouts.plantilla")

@section("cabecera")
     Actualizar un Departamento
@endsection

@section("cuerpo")
     <form          method="post"          action="/departamentos/{{
$departamento->id }}">
          @method("PUT")
          @csrf
          <table>
               <tr>
                    <td>Nombre</td>
```

```
                    <td><input type="text" name="nombre" value =
"{{ $departamento->nombre }}"></td>
                </tr>
                <tr>
                    <td>Presupuesto</td>
                    <td><input type="text" name="presupuesto"
value = "{{ $departamento->presupuesto }}"></td>
                </tr>
                <tr>
                    <td>Centro</td>
                    <td><select name="centro_id">
                        <option value="">Seleccione
Centro</option>
                        @foreach ($centros as $centro)
                            <option value="{{ $centro->id }}"
                            @if ($centro->id ==
$departamento->centro_id))
                            selected="selected"
                        @endif
                            >{{ $centro->nombre }}</option>
                        @endforeach
                    </select></td>
                </tr>
                <tr>
                    <td colspan="2" align="center"><input
type="submit" name="enviar" value="Enviar"></td>
                </tr>
            </table>
        </form>
        <form method="post" action="/departamentos/{{
$departamento->id }}">
            @method("DELETE")
            @csrf
            <input type="submit" name="borrar" value="Eliminar">
        </form>
        @if ($errors->any())
        <div>
            <ul>
                @foreach ($errors->all() as $error)
```

```
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
     @endif

@endsection

@section("pie")
@endsection \Illuminate\Http\Response
     */
    public function destroy($id)
    {
        //
        $centro = Centro::findOrFail($id);
        $centro->delete();

        return redirect('/centros');
    }
...
```

Anexos: Formas de personalizar los mensajes de validación.

En la validación de los formularios hemos personalizado los mensajes directamente en el controlador pero también se puede realizar a través del fichero de idiomas.

Si configuramos nuestra aplicación a español a través del fichero config.app

Fichero config app

```
'locale' => 'es',
```

En la carpeta r**esources/lang** tendremos que tener una carpeta “es” copia de la carpeta “en” que viene por defecto en Laravel. En esta carpeta “es” dentro del fichero validation.php pondremos las traducciones para que nos muestren los mensajes personalizados en español.

También podemos personalizar los mensajes creando nuestras propias reglas de validación. Para ello tendremos que crearnos un fichero de reglas con la siguiente instrucci

```
php artisan make:rule CentroFormRequest
```

Esto creará un nuevo fichero dentro de las carpetas HTT/request.
Lo primero que habrá que hacer en ese fichero es poner que el método **autorize** devuelva true

```
...
public function authorize()
{
    return true;
}
...
```

Ya sólo quedará definir los métodos para los atributos, reglas y mensajes que queramos personalizar.

Fichero de Regla para el campo nombre del Centro

```
<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class CentroFormRequest extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize()
    {
        return true;
    }

    public function attributes()
    {
        return [
```

```
            'nombre' => 'Nombre del centro de trabajo',
        ];
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules()
    {
        return [
            //
            'nombre' => 'required'
        ];
    }

    public function messages()
    {
        return [
            'nombre.required' => ':attribute no puede ser nulo',
        ];
    }
}
```

Si creamos nuestro propio fichero de reglas habrá que importarlo en el controlador para poder usarlo

```
...
use App\Http\Requests\CentroFormRequest;
...
```

Para método en el controlador no usaría la clase Request, en su lugar usaría CentroFormRequest, con lo que quedaría

```
/**
     * Store a newly created resource in storage.
     *
     * @param  \Illuminate\Http\Request  $request
```

```
     * @return \Illuminate\Http\Response
     */
    public function store(CentroFormRequest $request)
    {

        $centro = new Centro;
        $centro->nombre = $request->nombre;
        $centro->direccion = $request->direccion;
        $centro->save();


        return redirect('/centros');
    }
```

La información sobre validaciones en Laravel se puede consultar en el siguiente enlace:
Validaciones Laravel